Discrete event systems

Smedinga, Rein

Discrete Event Systems

Rein Smedinga
department of computing science
University of Groningen
p.o.box 800
9700 AV Groningen, the Netherlands
tel. +3150633937
E-mail: rein@cs.rug.nl

October 1993

Miraculous you call it babe
You ain't seen nothing yet
They've got Pepsi in the Andes
McDonalds in Tibet
Yosemite's been turned into
A golf course for the Japs
The Dead Sea is alive with rap
Between the Tigris and Euphrates
There's a leisure centre now
They've got all kinds of sports
They've got Bermuda shorts
They had sex in Pennsylvania
A Brazilian grew a tree
A doctor in Manhattan
Saved a dying man for free

It's a miracle
Another miracle
By the grace of God Almighty
And pressures of the market place
The human race has civilized itself
It's a miracle

Roger Waters — Amused to death (1992)

Preface

In this report we study discrete event systems in which qualitative aspects are of importance. Therefore, we introduce a model in which the order of the events is of sole importance. We use a triple consisting of the set of all possible events (the alphabet), the set of all behaviour (possible strings of events), and the set of all tasks (completed behaviour). We use this view to model synchronous as well as asynchronous connection of systems. Moreover, it is easy to define notions like control, deadlock, and livelock in this view.

Apart from a direct definition using the set of all possible behaviour and all possible tasks, we give a state space form of a system. This results in algorithms on automaton-like structures to compute all necessary subspaces and systems effectively. A program exists to perform this computations.[1]

This report is intended as textbook for the course on discrete event systems, given in the second trimester of the academic year 1993/1994. It also is a collection of the main results in the research on discrete event systems using trace theory as a basic approach.

Although the text is written with the outmost possible accuracy, no guarantee can be given on the correctness of the given results. Parts of this report also appeared as separate articles, e.q., see [Sme93b, Sme92].

[1] Information via email: `rein@cs.rug.nl`

Inhoudsopgave

Introduction

In discrete event systems we have to deal with discrete, asynchronous, and possibly non-deterministic actions. Discrete event systems appear in many different constitutions:

- sequential processes (doing a number of things in a row),
- concurrent processes (doing things concurrently, i.e., at the same time),
- operating systems (doing different things semi-parallel, i.e., via some form of interleaving),
- communication networks.

Different criteria can be considered for discussion. These criteria can be divided into:

- quantitative criteria (mostly performance measures like throughput, cycle time, and so on),
- qualitative criteria (absence of deadlock, no infinite loops, fairness).

For quantitative criteria networks of queues, (timed) Petri nets and alike can be used. We mention:

- pertubation analysis of queuing networks (developed by Y.C.Ho, see for example [HEC83], [HC83], [HC85] and [Cas93]),
- linear systems in the max algebra (see [CDQV83], [CDQV85], and [BCOQ92]).

Here, we are concerned with the qualitative criteria, i.e., to ensure some orderly flow of events. Modelling this flow of events can be done by:

- finite automata and formal languages (e.g., the supervisory control theory of Wonham, see [CDFV88], [IV88], [Ram83], [RW87], [WR87], [RW89]),
- Petri Nets (see [Pet81]),
- process algebra (e.g., the calculus of communicating systems (CCS) of Milner, see [Mil80], or the theory of communicating sequential processes (CSP), see [Hoa85]),
- temporal logic (see [TW86]),
- trace theory (as is done here).

Petri Nets can only be used to model one system, not to combine more systems. Both in CCS and CSP discrete processes are defined in a way very similar to the way we do here. Temporal logic is (merely) very difficult. No results have yet been found in the

direction of interaction of discrete events. Supervisory control theory comes closest to what we do here. However, it uses too much notation and is developed especially for control purposes, not for more general modelling.

In this report we study discrete event systems in which qualitative aspects are of importance. Such systems arise in the domains of manufacturing, computer and communication networks, robotics, vehicular traffic, and many others. Modelling such systems can be done at a logical level (consider only the logical order of the events), a temporal level (introducing time), or a stochastic level (introducing probabilities). Here, we consider the logical level.

Logical discrete event systems are first introduced by Hoare and Milner [Hoa85, Mil80]. Control of such systems is first introduced by Ramadge and Wonham ([RW87]), using a theory based on the theory of automata and languages.

The theory presented here is based on trace theory, introduced in [Sne85]. A logical discrete event system (DES for short) can then be denoted by two sets: one finite set of symbols representing the events and one set of finite sequences of such symbols, representing the behaviour.
Although trace theory was developed in the context of concurrent programs and found useful in modelling electronic components (see [Sne85, Udd84]), it seems to be the natural setting to model discrete events in general. In [Sme88] trace theory is first used to model discrete event systems, but without separated task and behaviour sets. In [Sme89] the ideas are used to define and solve control problems.

The approach as presented here differs form other modelling approaches in the sense that it does not impose any structure on the behaviour set, it just is a set of sequences of symbols. It turns out that, even with this simple definition, a DES can be defined, control problems can be formulated, etc. It is not even necessary (for example) to have a prefix closed behaviour.

We use a, for pure mathematicians perhaps nonstandard, alternative notation, developed by prof. E.W. Dijkstra, that leads to a clear, unambiguous way to display the theory. Also, proofs will be given in a different setting: instead of the (unfortunately too often used) mixture of English and bad mathematical notation we prefer this clearly mathematical style which does not lead to misinterpretations, vague arguments and difficult-to-find errors. We refer to [Sme93a] for a discussion on notation.

Notation

Throughout this paper the following notation[2] will be used:[3]

$(\forall x : B(x) : C(x))$ is true if $C(x)$ holds for every x satisfying $B(x)$, e.g.,
$(\forall x : x \in \mathbb{N} : x \geq 0)$
$(\exists x : B(x) : C(x))$ is true if there exists an x satisfying $B(x)$ for which $C(x)$ holds, e.g.,
$(\exists x : x \in \mathbb{N} : x \geq 10 \ \wedge \ x \leq 20)$
$(\exists! x : B(x) : C(x))$ is true if there exists exactly one such an x, e.g.,
$(\exists! x : x \in \mathbb{N} : x = 10)$

[2] Adapted from [Dij82]. See also [vG88] for a more thourough discussion on notation.
[3] The notation not only holds for single symbols x but also for tuples denoted by x.

$\{x : B(x) : y(x)\}$ is the set constructor and denotes the set of all elements $y(x)$ constructed using elements x satisfying $B(x)$, e.g.,[4]
$\{n : n \in \mathbb{N} : a^n b^n\} = \{\epsilon, ab, aabb, aaabbb, \ldots\}$

We will use the same notation for other quantifiers, like summation, maximum, minimum, etc, using the general form

$$(\mathit{quantifier} : \mathit{domain} : \mathit{result})$$

For equations (in proofs) we use the following notation:[2]

$$\begin{array}{ll} & A \\ = & \quad [\text{ hint why } A = B \text{ }] \\ & B \\ \Rightarrow & \quad [\text{ hint why } B \Rightarrow C \text{ }] \\ & C \end{array}$$

In the above sequence we have in fact written down the following:

$$((A = B) \ \wedge \ (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$$

If no hint is given, simple calculus is assumed.

[4] ϵ is the empty string
[2] See note 1 on previous page

1
Modelling discrete event systems

In this report we study logical discrete event systems. These are systems in which the order in occurrence of events is of importance, not the time of the occurrences. Such a system has a behaviour that can be described by giving all sequences of events that are possible. The number of such sequences can, of course, be infinite. We assume that events have no duration, i.e., they occur in infinitesimal time. This implies that no two events can ever occur at the very same time, except if these events *are* the same. In fact, we think of an event as a suddenly change in the state (whatever the state of a system might be). For example, if a person arrives at some point, we have such a suddenly (instantaneous) change in state: before the arrival n persons were present, after the arrival we have $n + 1$ persons. The arrival itself occurs in infinitesimal time: the one moment that person has not yet arrived, the next moment he is present.

This assumption is no restriction if we model a system by observing it. The system then is a black box and an observer writes down a symbol each time the corresponding event occurs. The behaviour of such a system is now given by all possible sequences of symbols (events) such an observer could write down.
A disadvantage of this approach is that we cannot distinguish between non-determinism and concurrency. See section 1.3.3.

1.1 Trace sets

The essence of a discrete event system lies in its behaviour, which is, in fact, a set of sequences of events. Events can be represented by letters, like a, b, or c, and sometimes also by words, like *arrival*, *stop*, or *enter*. A sequence of events then can be represented by a sequence of letters (or words) like $a \cdot b \cdot a \cdot c$, meaning that first event a occurs, then event b, then event a again, and, last, event c (see [Hoa85]). If we use simple letters for the events we will omit the concatenation operator $\cdot$ and write $abac$.

Such sequences of events are called traces. A special trace is the empty trace, denoted by ϵ. It represents the nothing-has-happened behaviour.

A number of operators on trace sets can be defined. We mention (with x and y traces, n some natural number, and A some set of events, called the alphabet, see [Sne85]):

concatenation	xy	first x, then y
choice	$x\|y$	x or y (but not both)

finite repetition	x^n	n times x: $\underbrace{x \cdots x}_{n \text{ times}}$
repetition	x^*	zero or more times x
non-empty repetition	x^+	one or more times x
weaving	x, y	shuffling of x and y (see further on)
restriction	$x \lceil A$	projection on alphabet a

Weaving introduces concurrency. If x and y have no events in common the trace x, y represents the concurrent behaviour of x and y. We will return to this subject later on (see example 1.12).
The alphabet restriction is defined by:

$$\begin{array}{lcll} \epsilon \lceil A & = & \epsilon & \\ xa \lceil A & = & x \lceil A & \text{if } a \notin A \\ & = & (x \lceil A)a & \text{if } a \in A \end{array}$$

Moreover, we have the operator **pref** that denotes the set of all prefices of a trace:

$$\mathbf{pref}(x) = \{y, z : yz = x : y\}$$

pref, $\lceil$, and * can also be performed on trace sets. If T is a trace set then

$$\begin{array}{lcl} \mathbf{pref}(T) & = & \{y, z : yz \in T : y\} \\ T \lceil A & = & \{x : x \in T : x \lceil A\} \end{array}$$

The alphabet generating set A^*, at last, is defined by

$$\epsilon \in A^*$$
$$x \in A^* \ \wedge \ a \in A \ \Rightarrow \ xa \in A^*$$

Sometimes we need the ordering of traces, the length of a trace, or the number of occurrences of some event in a trace. For two traces x and y the order of traces is defined by:

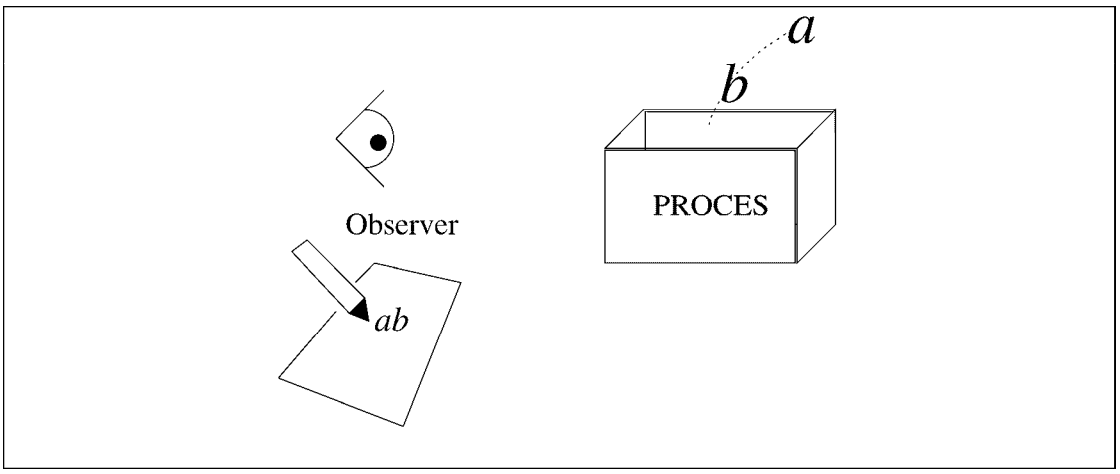
$$\begin{array}{lcl} x \leq y & \Leftrightarrow & x \in \mathbf{pref}(y) \\ x < y & \Leftrightarrow & x \leq y \ \wedge \ x \neq y \end{array}$$

For some trace x and event a the length of traces is defined by

$$\begin{array}{lcl} |\epsilon| & = & 0 \\ |xa| & = & |x| + 1 \end{array}$$

Last, for some trace x and events a and b the number of occurrences of a in x is denoted by $x \, \mathbf{N} \, a$ and defined by

$$\begin{array}{lcll} \epsilon \, \mathbf{N} \, a & = & 0 & \\ (xb) \, \mathbf{N} \, a & = & x \, \mathbf{N} \, a + 1 & \text{if } b = a \\ & = & x \, \mathbf{N} \, a & \text{if } b \neq a \end{array}$$



Figuur 1.1: The observer point of view. The observer has first seen event a happen and next event b. On his paper he has written down ab.

Example 1.1 To illustrate the above definitions:

$$
\begin{array}{lcl}
(ab)|(c^*d) & = & \{ab, d, cd, ccd, cccd, \ldots\} \\
ab^2cb\lceil\{a,c\} & = & ac \\
\mathbf{pref}(ab) & = & \{\epsilon, a, ab\} \\
\{a,b\}^* & = & \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\} \\
ab^2cb\,\mathbf{N}\,b & = & 3 \\
|ab^2cb| & = & 5 \\
ab < abc & &
\end{array}
$$

□

In appendix T more properties are mentioned concerning traces.

Exercise 1.1 Write out in full:

(a)	a^2b^3	(b)	$(a^*b)^2$
(c)	$(a^*b^*)^*$	(d)	ab, bc
(e)	ab, ba	(f)	ab, c
(g)	$abaab\lceil\{a\}$	(h)	$abaab\lceil\{c\}$
(i)	$\mathbf{pref}(aba)$	(j)	$\mathbf{pref}(a^*ba^*)$

□

1.2 A discrete event system

The simplest way to model a discrete event system is by giving all possible events and all (finite) sequences of events an observer could possibly see when observing the system. So:

$$P = \langle \mathbf{a}P, \mathbf{t}P \rangle$$

is such a system, where $\mathbf{a}P$ is the *alphabet*, the set of all possible events (and we suppose that we have only a finite number of different events) and $\mathbf{b}P$ is its *behaviour*, a subset of the set of all finite sequences of events over the alphabet $\mathbf{a}P$, so

$$\mathbf{b}P \subseteq (\mathbf{a}P)^*$$

Moreover, to get a system with a realistic interpretation, we add the restriction that if some sequence is in $\mathbf{b}P$, then also all prefices of that sequence should be in $\mathbf{b}P$. Formally, $\mathbf{b}P$ is *prefix-closed*:

$$\mathbf{pref}(\mathbf{b}P) = \mathbf{b}P$$

If $\mathbf{b}P = \emptyset$ we say the system is *empty* and if $\mathbf{b}P = (\mathbf{a}P)^*$ we say the system is *complete*. Notice that an empty system has no behaviour at all, not even the empty behaviour ϵ. If $\mathbf{b}P = \{\epsilon\}$ the system has precisely one behaviour: doing nothing.

Example 1.2 Mostly, we will simplify the system when we are modelling it in this way: Consider a system with a farmer, a wolf, a goat, and a cabbage in which the farmer should bring wolf, goat, and cabbage from one side of a river to the other. In order that the wolf does not eat the goat and the goat does not eat the cabbage, the farmer should be aware not to leave wolf and goat or goat and cabbage alone at either side of the river. In fact, this is some old puzzle: how should the farmer solve the problem when he is able to bring at most one item at the time to either side of the river.

Events we can think of are "farmer takes wolf from one side of the river to the other," "farmer takes goat from one side of the river to the other," "farmer takes cabbage from one side of the river to the other," and "farmer goes alone form one side of the river to the other." In fact, each such an event is a simplification of a great number of actions: "pick up the wolf," "put it into the boat," "go into the boat itself," "unrope," "row to the other side," etc. In modelling such a system we consider all these actions to be just one event (and therefore occurring at the same time, changing the state from "items are at some place" to "items are at the same place, except the wolf is now on the other side"). □

Modelling discrete event systems using $\langle \mathbf{a}P, \mathbf{t}P \rangle$ has one disadvantage: sometimes (mostly when considering more discrete event systems in cooperation with each other) we want to be able to tell when a system has ended legally or when it has ended illegally. A legal ending means that the system has performed a completed task. The above system of farmer, wolf, goat, and cabbage has ended legally if the farmer has succeeded in bringing wolf, goat, and cabbage to the other side of the river. Ending legally (performing a completed task) does not mean the system cannot continue: of course, after a legal behaviour more events may occur: the farmer may bring wolf, goat, and cabbage to this side of the river and afterwards to the other side again. After these actions he has, again, performed a completed task (the puzzle does not say anything about efficiency). A system can also end illegally. In that case no task is completed. If also no next event is possible the system has deadlocked.

To distinguish between legal and illegal ending we should add in the definition of a system all completed tasks, i.e., we define a discrete event system as a triple:

$$P = \langle \mathbf{a}P, \mathbf{b}P, \mathbf{t}P \rangle$$

with $\mathbf{a}P$ and $\mathbf{b}P$ as before and $\mathbf{t}P$ the task set of P with

$$\mathbf{t}P \subseteq (\mathbf{a}P)^*$$

Again, to get a realistic interpretation we add the restriction

$$\mathbf{t}P \subseteq \mathbf{b}P$$

(i.e., all tasks of P should belong to the behaviour of P). $\mathbf{t}P$ need not be prefix closed (as a completed task need in general not be complete if the last event of it has not yet occurred). Moreover, $\mathbf{b}P$ may be larger than the prefix closure of $\mathbf{t}P$ (i.e., in $\mathbf{b}P$ sequences of events may exist that are no prefix of a sequence of $\mathbf{t}P$). These extra sequences are called the locked sequences of the system, because once the system is in such a sequence no task can be completed. We return to this subject later on.
The restrictions

$$\mathbf{b}P = \mathbf{pref}(\mathbf{b}P) \qquad \mathbf{t}P \subseteq \mathbf{b}P$$

are added to get a system with a realistic interpretation. Such systems are called realistic DESs. Sometimes we will use systems in which these restrictions are not met. It may have, for example, a behaviour that is not prefix closed or a task that is no behaviour. Such systems go beyond our scope of a discrete event system. Nevertheless, they play a crucial role in some parts of the theory.
Summarized, we have

Definition 1.3 *A* discrete event system *(DES) P is defined by*

$$P = \langle \mathbf{a}P, \mathbf{b}P, \mathbf{t}P \rangle$$

with

$\mathbf{a}P$	*alphabet*	*finite set of events*
$\mathbf{b}P$	*behaviour*	$\mathbf{b}P \subseteq (\mathbf{a}P)^*$
$\mathbf{t}P$	*task set*	$\mathbf{t}P \subseteq (\mathbf{a}P)^*$

A realistic discrete event system *P is defined as a DES P with also*

prefix closed behaviour	$\mathbf{b}P = \mathbf{pref}(\mathbf{b}P)$
each task a behaviour	$\mathbf{t}P \subseteq \mathbf{b}P$

□

All theory developed below is based on such DESs. Of course, all derived properties also holds for realistic DESs.

Three DESs will be given a special name:

$$\begin{aligned} \mathbf{empty}(A) &= \langle A, \emptyset, \emptyset \rangle \\ \mathbf{skip}(A) &= \langle A, \{\epsilon\}, \{\epsilon\} \rangle \\ \mathbf{chaos}(A) &= \langle A, A^*, A^* \rangle \end{aligned}$$

If $\mathbf{b}P = \mathbf{pref}(\mathbf{t}P)$ we say the system is *lock free*. Lock free systems will be denoted by a tuple $\langle \mathbf{a}P, \mathbf{t}P \rangle$, where $\mathbf{b}P$ is left out because it can easily be determined from $\mathbf{t}P$. The term lock free corresponds with the term *nonblocking* as is used in supervisory theory (see [RW89]).

Exercise 1.2 Proof that a lock free system is realistic. □

Example 1.4 A one-place-buffer can be modelled as the following lock free system:

$$\mathit{buffer} = \langle \{in, out\}, (in \cdot out)^* \rangle$$

□

1.2.1 Ordering of systems

The ordering of discrete event systems is defined only for systems with the same alphabet as:

$$P \subseteq R$$
$$=$$
$$\mathbf{a}P = \mathbf{a}R \ \wedge \ \mathbf{t}P \subseteq \mathbf{t}R \ \wedge \ \mathbf{b}P \subseteq \mathbf{b}R$$

If $P \subseteq R$ we say that P is a *subsystem* of R.

Exercise 1.3 Proof:

$$(\forall P :: \mathbf{empty}(aP) \subseteq P \subseteq \mathbf{chaos}(\mathbf{a}P))$$

□

1.2.2 Alphabet restriction on DESs

Alphabet restriction has been defined on separate traces and also on trace sets before. It can easily be extented on DESs as well:

$$P\lceil A = \langle \mathbf{a}P \cap A, \mathbf{b}P\lceil A, \mathbf{t}P\lceil A \rangle$$

Exercise 1.4 Given are

$$P_1 = \langle \{a,b\}, a^*|b^+, a^+ \rangle \qquad P_2 = \langle \{a,b,c\}, a^*b^*, a^*b^* \rangle$$
$$P_3 = \langle \{a,b\}, ab, b \rangle \qquad P_4 = \langle \{a,b\}, a^*b, a|b \rangle$$

Check if $P_1 \subseteq P_2$ and if $P_3 \subseteq P_4$.
Which of these systems are realistic?
Which of these systems are lock free? □

1.2.3 The realistic interior of a DES

Sometimes it is useful to be able to get the greatest subsystem of a DES that is realistic. Therefore we have the so called realistic interior:

Definition 1.5 *The* realistic interior *of some DES P is defined by*

$$\mathbf{real}(P) = \langle \ \mathbf{a}P,$$
$$\{x : x \in \mathbf{b}P \ \wedge \ (\forall y : y \leq x : y \in \mathbf{b}P) : x\},$$
$$\{x : x \in \mathbf{t}P \ \wedge \ (\forall y : y \leq x : y \in \mathbf{b}P) : x\} \rangle$$

□

Notice that $\mathbf{t}(\mathbf{real}(P)) = \mathbf{t}P \cap \mathbf{b}(\mathbf{real}(P))$. We have the following properties:

Property 1.6

(*a*) $\mathbf{real}(P) \subseteq P$
(*b*) $\mathbf{real}(P)$ *is a realistic DES*
(*c*) $\mathbf{real}(P)$ *is the greatest realistic DES that is a subsystem of* P

□

Exercise 1.5 Compute the realistic interior of $\langle \{a,b\}, (a^*|b^+a^*), (ab) \rangle$. □

1.3 Reflection of a DES

As in set theory we will give a definition of the complement of a DES, sometimes called a *dual system*. It has as behaviour every trace that is no behaviour of the original system and as task set every trace that is no task in the original system. This dual system can be found using the reflection operator.

1.3.1 The reflection operator

From [T.V90] and [T.V91] we have the following definition of the dual system of some DES:

Definition 1.7 *The* dual system *of a DES P is defined by*

$$\sim P = \langle \mathbf{a}P,\ (\mathbf{a}P)^* \backslash \mathbf{b}P,\ (\mathbf{a}P)^* \backslash \mathbf{t}P \rangle$$

□

Notice that, if P is a realistic DES, $\sim P$ in general is not. Because the reflection operator will play a crucial role in controlling a realistic DES and leads to probably unrealistic DESs, this is why we need unrealistic DESs as well.

Property 1.8

(a) $P \subseteq R \Leftrightarrow \sim R \subseteq \sim P$
(b) $\sim\sim P = P$
(c) $\sim\mathbf{skip}(\emptyset) = \mathbf{empty}(\emptyset)$
(d) $\sim\mathbf{chaos}(A) = \mathbf{empty}(A)$

□

Exercise 1.6 Compute $\sim(\langle\{a,b\}, \mathbf{pref}(a|b), (a|b)\rangle)$ and $\sim(\langle\{a\}, (a|aa), (a)\rangle)$ and their realistic subsystems. □

1.3.2 Relation to general dynamical systems

In [Wil88] a general definition is given of a dynamical system $\Sigma = (T, W, B)$ with

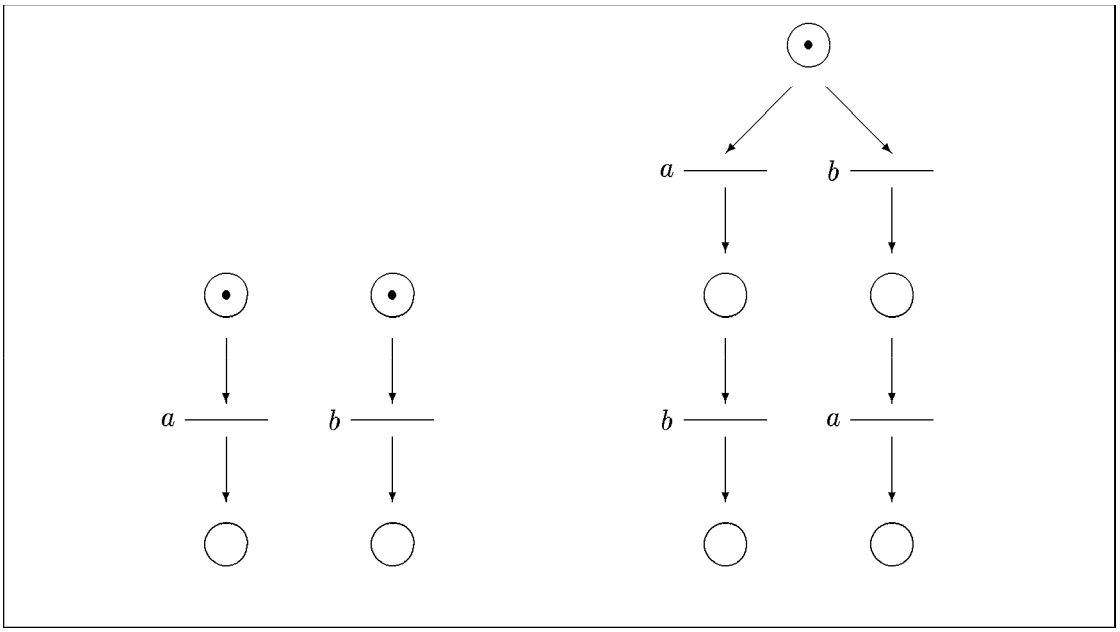
T	time axis (normally $T \subseteq \mathbb{R}$ or $T \subseteq \mathbb{Z}$)
W	signal alphabet (some abstract set)
$B \subseteq W^T$	the behaviour

This definition is (to quote Jan Willems) "hopelessly general but nevertheless it captures rather well the crucial features of the notion of a dynamical system." For example, an n-dimensional continuous system can be modelled this way with $W \subseteq \mathbb{R}^n$ and $B \subseteq (\mathbb{R}^n)^T$ a set of n-dimensional functions of t satisfying some (physical) laws and describing the system.

A discrete event system P is a special kind of a dynamical system Σ with[1]

$$\begin{array}{lcl} T & = & \mathbb{N} \\ W & = & \mathbf{a}P \cup \{\Box\} \\ B & = & \{w : (\exists t_e : t_e \in \mathbb{N} : w[0..t_e) \in \mathbf{b}P \ \wedge\ (\forall t : t \geq t_e : w(t) = \Box)) : w\} \end{array}$$

[1] $w[0..t_e)$ stands for the string $w[0]w[1]\ldots w[t_e - 1]$.



Figuur 1.2: A Petri-Net model for a system with concurrency of events a and b and a model for a system with non-deterministic choice between ab and ba

The blanks ($\Box$) are needed for cosmetic reasons: B contains only infinite strings, while $\mathbf{b}P$ contains only finite strings, which have to be completed by adding blanks at the end to fit the definition. The time index t is interpreted here as logic time (i.e., it parametrizes the order of events) and not as clock time, which is the usual case in conventional system theory.

The task set of P does not occur in this comparison. It is a special property of the system.

1.3.3 Concurrency and non-deterministic choice

A DES as defined here describes in fact the behaviour that can be observed. If we look at such a system from a distance and write down each event that occurs we get a trace, a behaviour of that system. Describing the system in this way has one disadvantage. We cannot tell the difference between a system that is able to perform two events a and b in parallel and a system that can choose between behaviour "first a and then b" and "first b and then a." Because of the fact that events are assumed to be infinitesimal in time (due to the fact that an observer cannot write down the names of more then one event at the same time) we will observe for both systems first the occurrence of a and then the occurrence of b or first b and then a. From this two observations we cannot tell if a and b are meant to be done in parallel or if there is some non-deterministic choice.

As long as observational behaviour is of importance it does not matter that this difference cannot be seen. In fact, only in Petri-Nets, see [Pet81], the difference can be modelled. In figure 1.2 both systems are displayed.

1.4 More discrete event systems

Now we know how to model one discrete event system we could investigate the cooperation of more discrete event systems. In fact, at the beginning of this chapter, we have already mentioned the important assumption that no two events can ever occur at the very same time, unless those events are the same. From this assumption we might conclude that different discrete event systems should cooperate via common events, events that are the same in these systems (see [Hoa85, Sne85]).

But how could one event be the same in, say, two different discrete event systems? Consider for a moment a vending machine[2] that gives coffee after inserting a coin. The machine can be modelled with task set equal to

$$(\mathit{coin} \cdot \mathit{coffee})^*$$

The event *coin* represents the insertion of the coin by a person as well as the acceptance of the coin by the vending machine. Also, the event *coffee* represents producing the coffee by the machine as well as accepting the coffee by the person. Both events are representations of a number of actions, partly performed by the vending machine, partly by the person who wants coffee. The events are common to both vending machine and person and therefore only occur if both systems can engage in it: the vending machine must be able to accept a coin and the person must be able to insert it in order for event *coin* to occur. In this example no coffee can be produced by the vending machine if the person wants tea instead of coffee.

The event *coin* in the discrete event system "vending machine" only occurs if the event *coin* in the discrete event system "person" occurs also (and at the same time). This kind of interaction is called *synchronous interaction* (see [Hoa85, Sne85]): a common event only occurs if all systems involved can engage in it.

1.5 Synchronous connection

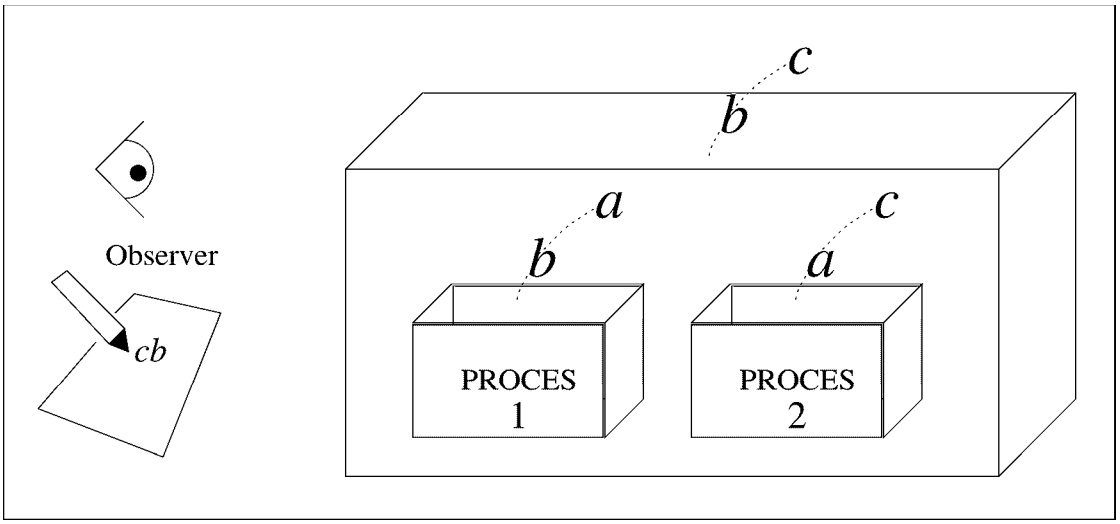
There is another way of looking at (synchronous) interaction of two systems. We also can say that the resulting behaviour of these two systems should be such that, if we restrict this behaviour to the alphabet of one of the systems, we get a behaviour from that system. This way of explaining interaction is equivalent to the previous one, but it is simpler to make formal. Therefore, the definition of interaction of discrete event systems makes use of this view:

Definition 1.9 *The system that results if two discrete event systems P and R are interacted is called the* connection *of P and R, denoted by $P \parallel R$, and defined by*

$$\begin{aligned} & P \parallel R \\ = \\ & \langle \mathbf{a}P \cup \mathbf{a}R \ , \{x : x \in (\mathbf{a}P \cup \mathbf{a}R)^* \ \wedge \ x\lceil \mathbf{a}P \in \mathbf{b}P \ \wedge \ x\lceil \mathbf{a}R \in \mathbf{b}R : x\} \\ & \qquad\qquad , \{x : x \in (\mathbf{a}P \cup \mathbf{a}R)^* \ \wedge \ x\lceil \mathbf{a}P \in \mathbf{t}P \ \wedge \ x\lceil \mathbf{a}R \in \mathbf{t}R : x\}\rangle \end{aligned}$$

□

[2] The example of a vending machine is used often in literature. One way or the other vending machines illustrate best the cooperation between two systems: the vending machine at one hand, a (thirsty) person at the other.



Figuur 1.3: The observer view for a connection of two systems. Process 1 has done ab and process 2 ca. If the observer cannot see internal events, its observation is cb, and this is on his piece of paper.

The operator $\|$, the connector, arises from trace theory (see [Sne85]), were it is called the *weave*-operator and denoted by $\mathbf{w}$. It is a shuffle, were common events occur simultaneously.

Example 1.10 Consider the lock free systems:

$$P = \langle \{a,b,c,d\}, (abc|ad) \rangle$$
$$R = \langle \{a,c\}, (ac) \rangle$$

then we have

$$P \,\|\, R \;=\; \langle \{a,b,c,d\}, \mathbf{pref}(ad|abc), (abc) \rangle$$

for example: $abc \in \mathbf{t}(P \,\|\, R)$ because $abc|\mathbf{a}P = abc \in \mathbf{t}P$ and $abc|\mathbf{a}R = ac \in \mathbf{t}R$. □

Example 1.11 For separate traces we can use the comma-operator to denote weaving (notice that concatenation has higher priority then weaving and choice):

$$(ab, bc) \;=\; (abc)$$
$$(ab, cb) \;=\; (acb|cab)$$

□

Example 1.12 The operator $\|$ denotes not only synchronization but also concurrency. Events that are not common may occur concurrently. If $x = ab$ and $y = de$ then x, y denotes the concurrent behaviour of ab and de, i.e.,

$$(x, y) = \{abde, adbe, adeb, dabe, daeb, deab\}$$

Because of our assumption that any event occurs in infinitesimal time no two different events can really occur at the very same time, but always in some order. If that order is arbitrary, we have concurrency (see section 1.3.3). □

Exercise 1.7 Compute:

$\langle\{a,b,c\}, \mathbf{pref}(abc), (ab)\rangle \parallel \langle\{a,b\}, \mathbf{pref}(ab), (a)\rangle$
$\langle\{a,b,c\}, \mathbf{pref}(ab), (ab)\rangle \parallel \langle\{c\}, (c), (c)\rangle$
$\langle\{a,b\}, \mathbf{pref}(ab), (ab)\rangle \parallel \langle\{c\}, (c), (c)\rangle$ □

1.5.1 Properties of the connection operator

The operator $\parallel$ defines a binary operation on the set of discrete event systems. It has some nice properties, which are listed below:

Property 1.13 *For general systems P, R, and S, the following hold:*

(a)	*symmetry:*	$P \parallel R = R \parallel P$
(b)	*unit element:*	$P \parallel \mathbf{skip}(\emptyset) = P$
(c)	*the zero element:*[3]	$P \parallel \mathbf{empty}(\emptyset) = \mathbf{empty}(\mathbf{a}P)$
(d)	*idempotenty:*	$P \parallel P = P$
(e)	*associativety:*	$(P \parallel R) \parallel S = P \parallel (R \parallel S)$

□

1.5.2 Synchronous external connection

Sometimes we are no longer interested in the events that are common to P and R in the connection $P \parallel R$, because they are used for this connection and are no longer free. Then we use the external connection.

Definition 1.14 *The* external connection *of two systems P and R is defined by*[4]

$$P \rceil\lceil R = (P \parallel R)\lceil(\mathbf{a}P \div \mathbf{a}R)$$

❑

The operator $\rceil\lceil$ is like the *blend* in trace theory (see [Sne85] and appendix T). It has the following similar properties:

Property 1.15 *For general systems P, R, and S, the following hold:*

(a)	*symmetry:*	$P \rceil\lceil R = R \rceil\lceil P$
(b)	*unit element:*	$P \rceil\lceil \mathbf{skip}(\emptyset) = P$
(c)	*zero element:*	$P \rceil\lceil \mathbf{empty}(\emptyset) = \mathbf{empty}(\mathbf{a}P\rangle$
(d)	*nonidempotenty:*	$P \neq \mathbf{empty}(\emptyset) \Rightarrow P \rceil\lceil P = \mathbf{skip}(\emptyset)$

□

The operator is, in general, not associative. However, we have

Property 1.16 *If no event occurs in more than two of the alphabets, the operator $\rceil\lceil$ is associative.* □

Exercise 1.8 Repeat exercise 1.7 with $\parallel$ replaced by $\rceil\lceil$. □

The following property relates ordering to connection:

[3] We do not have $P \parallel \mathbf{empty}(\emptyset) = \mathbf{empty}(\emptyset)$, so $\mathbf{empty}(\emptyset)$ is not really a zero, but it reduces all behaviour to nothing, so the name is not completely misplaced here.

[4] $A \div B$ denotes the symmetric set difference, i.e., $A \div B = (A \cup B) \backslash (A \cap B)$.

Property 1.17 *For discrete event systems P, R_1, and R_2 with $\mathbf{a}R_1 = \mathbf{a}R_2$, we have:*

(a) $R_1 \subseteq R_2 \Rightarrow (P \parallel R_1) \subseteq (P \parallel R_2)$
(b) $R_1 \subseteq R_2 \Rightarrow (P \rceil\lceil R_1) \subseteq (P \rceil\lceil R_2)$

□

And the following property relates connection to reflection:

Property 1.18

(a) $P \parallel \sim P = \mathbf{empty}(\mathbf{a}P)$
(b) $P \rceil\lceil \sim P = \mathbf{empty}(\emptyset)$

□

1.5.3 More operators on systems

We use set operators on discrete event systems as well, using the following definition:

Definition 1.19 *Given two discrete event systems P and R, then we define the* union *of P and R, denoted $P \cup R$ (pronounce "P or R"), by*

$$\langle \mathbf{a}P \cup \mathbf{a}R, \mathbf{b}P \cup \mathbf{b}R, \mathbf{t}P \cup \mathbf{t}R \rangle$$

and the intersection *of P and R, denoted $P \cap R$ (pronounce "P and R"), by*

$$\langle \mathbf{a}P \cap \mathbf{a}R, \mathbf{b}P \cap \mathbf{b}R, \mathbf{t}P \cap \mathbf{t}R \rangle$$

and, if $\mathbf{a}P = \mathbf{a}R$, the exclusion *of P and R, denoted $P \backslash R$ (pronounce "P without R"), by*

$$\langle \mathbf{a}P, \mathbf{b}P \backslash \mathbf{b}R, \mathbf{t}P \backslash \mathbf{t}R \rangle$$

□

Exercise 1.9 Compute:

$\langle \{a,b,c\}, \mathbf{pref}(ab), (ab) \rangle \cup \langle \{a\}, a^*, a \rangle$
$\langle \{a,b,c\}, \mathbf{pref}(ab), (ab) \rangle \cap \langle \{a\}, a^*, a \rangle$
$\langle \{a,b,c\}, (ab)^*, (ab)^+ \rangle \backslash \langle \{a,b,c\}, (ab)^+, (ab)^* \rangle$

□

Property 1.20 *For discrete event systems P and R, we have:*

(a) $\mathbf{a}P = \mathbf{a}R \;\Rightarrow\; (P \parallel R) = P \cap R$
(b) $\mathbf{a}R \subseteq \mathbf{a}P \;\Rightarrow\; (P \parallel R) \lceil \mathbf{a}R = (P \lceil \mathbf{a}R) \cap R$

proof: see property T.14 □

Property 1.21 *For discrete event systems P and R and alphabet A, we have:*

(a) $(P \parallel R)\lceil A \quad \subseteq \quad (P\lceil A) \parallel (R\lceil A)$
(b) $\mathbf{a}P \cap \mathbf{a}R \subseteq A \;\Rightarrow\; (P \parallel R)\lceil A = (P\lceil A) \parallel (R\lceil A)$
(c) $(P \rceil\lceil R)\lceil A \quad \subseteq \quad (P\lceil A) \rceil\lceil (R\lceil A)$
(d) $\mathbf{a}P \cap \mathbf{a}R \subseteq A \;\Rightarrow\; (P \rceil\lceil R)\lceil A = (P\lceil A) \rceil\lceil (R\lceil A)$

proof: see property T.15 □

Corollary 1.22

$$(P_1 \parallel P_2)\lceil \mathbf{a}P_1 \subseteq P_1$$

proof: See corollary T.16 □

Property 1.23 *For systems P, R, and S, with $\mathbf{a}P = \mathbf{a}R$, we have:*

$$\begin{array}{lrcl}
(a) & S \parallel (P \cup R) & = & (S \parallel P) \cup (S \parallel P) \\
(b) & S \parallel (P \cap R) & = & (S \parallel P) \cap (S \parallel P) \\
(c) & S \parallel (P \backslash R) & = & (S \parallel P) \backslash (S \parallel R) \\
(d) & S \rceil\lceil (P \cup R) & = & (S \rceil\lceil P) \cup (S \rceil\lceil P) \\
(e) & S \rceil\parallel (P \cap R) & \subseteq & (S \rceil\parallel P) \cap (S \rceil\parallel P)
\end{array}$$

proof: See property T.17 □

Notice the $\subseteq$ in part (e) of this property.

Property 1.24 *For DESs P and R with $\mathbf{a}P = \mathbf{a}R$ we have:*

$$\begin{array}{ll}
(a) & \sim(P \cup R) = \sim P \cap \sim R \\
(b) & \sim(P \cap R) = \sim P \cup \sim R
\end{array}$$

proof: Take $A = \mathbf{a}P = \mathbf{a}R$, then:

$$\begin{array}{ll}
 & \sim(P \cup R) \\
= & \quad [\text{ definition of } \cup\ ] \\
 & \sim\langle A, \mathbf{b}P \cup \mathbf{b}R, \mathbf{t}P \cup \mathbf{t}R\rangle \\
= & \quad [\text{ definition of } \sim\ ] \\
 & \langle A, A^* \backslash (\mathbf{b}P \cup \mathbf{b}R), A^* \backslash (\mathbf{t}P \cup \mathbf{t}R)\rangle \\
= & \quad [\ X \backslash (B \cup C) = X \backslash B \cap X \backslash C\ ] \\
 & \langle A, A^* \backslash (\mathbf{b}P) \cap A^* \backslash (\mathbf{b}R), A^* \backslash (\mathbf{t}P) \cap A^* \backslash (\mathbf{t}R)\rangle \\
= & \quad [\text{ definition of } \cap\ ] \\
 & \langle A, A^* \backslash (\mathbf{b}P), A^* \backslash (\mathbf{t}P)\rangle \cap \langle A, A^* \backslash (\mathbf{b}R), A^* \backslash (\mathbf{t}R)\rangle \\
= & \quad [\text{ definition of } \sim\ ] \\
 & \sim\langle A, \mathbf{b}P, \mathbf{t}P\rangle \cap \sim\langle A, \mathbf{b}R, \mathbf{t}R\rangle \\
= & \\
 & \sim P \cap \sim R
\end{array}$$

(b) is similar. □

Property 1.25

$$A_1 \cup A_2 = \mathbf{a}P \;\Rightarrow\; P \subseteq P\lceil A_1 \parallel P\lceil A_2$$

❑

Exercise 1.10 Show that in general $P \neq P\lceil A_1 \parallel P\lceil A_2$ □

Exercise 1.11 Proof that for $A = \mathbf{a}P \cup \mathbf{a}R$: $P \parallel R = (P \parallel A^*) \cap (R \parallel A^*)$. □

Property 1.26 *For systems P and R and alphabet A, we have:*

$$\begin{array}{lllll} (a) & (P\lceil A)\backslash(R\lceil A) & \subseteq & (P\backslash R)\lceil A \\ (b) & (P \cup R)\lceil A & = & (P\lceil A) \cup (R\lceil A) \\ (c) & (P \cap R)\lceil A & \subseteq & (P\lceil A) \cap (R\lceil A) \end{array}$$

proof: See property T.18 □

We do not have equality for (a):

Example 1.27 Consider $A = \{a\}$ and

$$P = \langle \{a, b, c\}, (ab|ac) \rangle$$
$$R = \langle \{a, b, c\}, (ab|bc) \rangle$$

then

$$P\lceil A = \langle \{a\}, (a) \rangle$$
$$R\lceil A = \langle \{a\}, (a) \rangle$$
$$(P\backslash R)\lceil A = \langle \{a, b, c, \}, (ac) \rangle \lceil A = \langle \{a\}, (a) \rangle$$
$$(P\lceil A)\backslash(R\lceil A) = \langle \{a\}, \emptyset \rangle$$

□

In [Sne85] it is emphasized that it is essential to involve the alphabet into the definition of connection in order for $\|$ to be associative. Because of this associative property we can connect more discrete event systems without worrying about the order of computation.

Exercise 1.12 To show the above, compute both:

$$((ba|\epsilon) \,\|\, (ab|\epsilon)) \,\|\, (ab|\epsilon) \qquad \text{and} \qquad (ba|\epsilon) \,\|\, ((ab|\epsilon) \,\|\, (ab|\epsilon))$$

if the alphabets are implicit and show that associativity for $\|$ fails. □

We use the following notation for connection of more discrete event systems (here $\mathcal{P}$ stands for some set of DESs):

$$\begin{array}{lll} (\|\, P : P \in \emptyset : P) & = & \mathbf{skip}(\emptyset) \\ (\|\, P : P \in (\{R\} \cup \mathcal{P}) : P) & = & R \,\|\, (\|\, P : P \in \mathcal{P} : P) \end{array}$$

If indices are involved, we sometimes use

$$(\|\, i : i \in I : P_i)$$

with the corresponding meaning.

1.6 A directed connection

Using the above definition of connection means dealing with synchronous interaction. Events have no direction: sending and receiving occurs at the same time. We also have *asynchronous interaction*: events then have a direction: sending goes before receiving. In that case inserting a coin by a person should come before accepting it by the vending machine and producing coffee should go before accepting it. In that case "sending" and "receiving" are different events (although related to each other). We can distinguish between sending (inserting a coin) and receiving (accepting it) by postfixing the event by ! and ? respectively. E.g., *coin*! is inserting a coin, *coin*? is accepting it, *coffee*! is producing coffee, *coffee*? is accepting it. Asynchronous interaction now means that "receiving should come after sending," so *coffee*? should come after *coffee*!.

Asynchronous interaction can be defined using the synchronous interaction operator $\|$. In order to do so we should first devide the events into two kinds: inputs $\imath P$ and outputs $\mathbf{o}P$, so that

$$\mathbf{a}P = \imath P \cup \mathbf{o}P \qquad \imath P \cap \mathbf{o}P = \emptyset$$

A *(directed) discrete event system* therefore is a discrete event system with all events in $\mathbf{o}P$ postfixed by ! and all events in $\imath P$ postfixed by ?. Such a discrete event system is denoted by $P?!$. The (directed) task set of the vending machine then equals

$$(\mathit{coin}? \cdot \mathit{coffee}!)^*$$

and the (directed) task set of the person equals

$$(\mathit{coin}! \cdot \mathit{coffee}?)^*$$

Connection can now be defined with the use of the following additional set and system:

$$\begin{array}{lcl} \mathcal{P} & = & \{a : a \in A : \langle (a! \cdot a?)^*, \{a!, a?\}\rangle\} \\ \mathbf{trans}(A) & = & (\|\, P : P \in \mathcal{P} : P) \end{array}$$

P is the set of all lock free discrete event systems with alphabet equal to $\{a!, a?\}$ for some $a \in A$ and task set equal to $(a!a?)^*$, i.e., first sending and then receiving. The system **trans** is the shuffle (or parallel composition) of all these discrete event systems. **trans** denotes the transmission of events from alphabet A: its tasks set equals all sequences in which for each event from A its sending part and its receiving part occurs equally often and alternatingly (first sending, then receiving). So

$$\mathit{coin}! \cdot \mathit{coin}? \cdot \mathit{coffee}! \cdot \mathit{coin}! \cdot \mathit{coin}? \cdot \mathit{coffee}?$$

belongs to the task set of $\mathbf{trans}(\{\mathit{coin}\ ,\mathit{coffee}\ \})$ but

$$\mathit{coin}! \cdot \mathit{coin}? \cdot \mathit{coffee}!$$
$$\mathit{coin}! \cdot \mathit{coin}! \cdot \mathit{coffee}! \cdot \mathit{coin}? \cdot \mathit{coin}?$$

do not (the first, however, does belong to the behaviour).

Definition 1.28 *The* directed connection *of P and R is denoted by $P \overleftrightarrow{\|} R$ and defined only if $\mathbf{o}P \cap \mathbf{o}R = \imath P \cap \imath R = \emptyset$ by*

$$P \overleftrightarrow{\|} R = P?! \,\|\, \mathbf{trans}(\mathbf{a}P \cap \mathbf{a}R) \,\|\, R?!$$

□

Example 1.29 For our vending machine example (with V the vending machine and P the person) we find:

$$\begin{array}{lcl} \mathbf{t}P?! & = & (coin! \cdot coffee?)^* \\ \mathbf{t}V?! & = & (coin? \cdot coffee!)^* \\ \mathbf{t}(\mathbf{trans}(\{coin, coffee\}) & = & ((coin! \cdot coin?)^*, (coffee! \cdot coffee?)^*) \end{array}$$

which results in

$$\mathbf{t}(P \overleftrightarrow{\parallel} V) = (coin! \cdot coin? \cdot coffee! \cdot coffee?)^*$$

□

In **trans** we have modelled that each output should first be followed by the corresponding input before that output may occur again (no insertion of a next coin if the previous one is not yet accepted). $\mathbf{trans}(A)$ is in fact some buffer: it may hold precisely one output event for each event in A. It models *bounded delay* (each output should first be followed by the corresponding input before it may occur again) and *overtaking* (different outputs are not necessarily followed by the corresponding inputs in the same order).

trans can also be defined such that more outputs may take place before a corresponding input has occurred. If the number of occurrences of outputs that has not yet been received is infinite we speak of unbounded delay. In that case **trans** is defined using:

$$\begin{array}{l} \mathcal{P}(A) \\ = \\ \{a : a \in A : \langle \{a!, a?\}, \{x : x \in \{a!, a?\}^* \ \wedge \ (\forall y : y \leq x : y \,\mathbf{N}\, a! \geq y \,\mathbf{N}\, a?) : x\}\rangle\} \end{array}$$

An operator in trace theory exists that takes care of this unbounded delay. It is the *agglutinate* and can be found in [Sne85]. Its disadvantage is its difficulty and (more important) it looses some important properties (regularity for example), which makes the operator unusable in state space form (see next chapter).

Exercise 1.13 Consider

$$\begin{array}{llll} P = \langle \{a, b, c\}, (a|b|c)^* \rangle & & R = \langle \{a, b, c\}, (abc) \rangle & \\ {}_1P = \{a, b\} & \mathbf{o}P = \{c\} & {}_1R = \{c\} & \mathbf{o}R = \{a, b\} \end{array}$$

Give $P!?$ and $R!?$ and compute $P \overleftrightarrow{\parallel} R$. □

References

Modelling discrete event systems in a straightforward sense as is done in this article originally comes from [Hoa85], although Hoare uses recurrent equations, which is only one way of defining the behaviour. The ideas of synchronizing discrete event systems using common events that should occur in all systems that are involved at the same moment is also first stated in [Hoa85] and [Mil80].

Discrete event systems are also studied by Wonham and Ramadge, for an overview see [RW89]. They use a state space to model the behaviour of a system. Although the synchronization of systems looks alike there are some (minor) differences: system and

controller (supervisor) do not act precisely the same: the controller follows the system and control depends on the state the controller is in. As in this report, Ramadge and Wonham also deal with two aspects of systems: they have $L(G)$, the set of all traces that can be generated by the graph of the system, corresponding to the behaviour of the system, and $L_{\mathrm{m}}(G)$, the set of all traces that end in a marker state, corresponding to the task set in our definition. In the approach of Ramadge and Wonham (see [RW87, RW89] a realistic DES is modelled using a deterministic automaton. The resulting languages generated by the automaton correspond to the behaviour and task set in our approach. However, our approach does not need a representation of the behaviour and task set. It works on the sets itself. Instead of an automaton also other representations can be used to model the system. We mention the well-known command-structure from trace theory (see [Kal88, Sne85]) and the recurrent expressions from [Hoa85]. The theory derived in this report does not assume any modelling of behaviour and task sets.

In the supervisory theory of Ramadge and Wonham the above defined interaction of two systems is only part of the connection of a system (a plant) and its controller. In their approach the controller, dependent on the state he is in, can enable or disable events in the plant. Enabling or disabling in our approach is implicit. If the system is in a state where it is able to do some (common) event a and the controller is in a state where a is unable, a is disabled in the connection. Notice that this form of disabling events is fully symmetrical with respect to plant and controller.

Moreover, we do not distinguish between controllable and uncontrollable events beforehand. Again, this is implicit. If we chose a controller R with alphabet $\mathbf{a}R$, then $\mathbf{a}R$ is the set of controllable events. However, $\mathbf{a}R$ may also contain events that are not in $\mathbf{a}P$.

In our approach a controller is also a system. In the supervisory theory a controller is a function $f\colon \mathbf{b}P \to \Gamma$, giving, for each behaviour of P a control input to be applied on P, by which events are disabled.

Our approach also deals with partial observations: only those events that are common to plant and controller can be observed by the controller. No additional observation-alphabet and projection or mask is needed to model partial observability as has to be done in the supervisory approach (see [RW89]).

Another set-up for discrete event systems can be found in [IV88]. The modelling is much more difficult (it also models infinity long sequences), but does not, at the moment, lead to more useful tools.

This chapter is part of [Sme93b].

2

A state space form for discrete event systems

The states of a discrete event system can be made visible using diagrams. From one state to another arrows can be drawn to represent the occurrence of an event (the arrow is then labelled with the name of that event). Each path through such a diagram then is a behaviour of the system.

2.1 State graphs

It is well-known that we can associate with a trace structure a (finite) state automaton. Each path in the automaton, starting in the initial state and ending in a final state corresponds to a trace in the trace set. If the trace structure is regular, the number of needed states is finite. A DES is in fact a pair of trace structures, so we could use two automatons to represent one DES. However, in this way we lose the correspondence between behaviour and task. Instead, we use a more general automaton, called a state graph containing in fact two kinds of final states.

Definition 2.1 *A* state graph G *is defined by*

$$G = (A, Q, \delta, q, B, T)$$

with

A	*alphabet, a finite set of labels*
Q	*the state set*
δ	*the state transition function:* $\delta: Q \times A \rightarrow Q$
q	*the initial state:* $q \in Q$
B	*the behaviour state set:* $B \subseteq Q$
T	*the task state set:* $T \subseteq Q$

□

The alphabet corresponds to the alphabet of the discrete system it represents (and so should also be finite). The state set is some (presumably infinite) set of states. One state q is the initial state (the diagram starts there, it represents the initial state the discrete system is in). The state transition function represents the occurrences of the events. $\delta(p, a)$ denotes the state the system will be in if it is in state p and event a occurs. We assume that δ is a total function. Moreover, we assume that Q is never empty, i.e., contains at least the initial state q.

The subsets B and T correspond to the behaviour and task set of the system it represent. The exact correspondence will be explained in the following two sections.

We can consider all possible paths through this diagram using the *closure* of the transition function $\delta^*: Q \times A^* \rightarrow Q$, defined by

$$\begin{array}{lcl} \delta^*(p, \epsilon) & = & p \\ \delta^*(p, xa) & = & \delta(\delta^*(p, x), a) \quad a \in A, x \in A^* \end{array}$$

2.1.1 From DES to state graph

For a DES P we can construct a state graph using the following equivalence classes:

$$[x]_P = \{y : (\forall z :: xz \in \mathbf{b}P \Leftrightarrow yz \in \mathbf{b}P) \ \wedge \ (\forall z :: xz \in \mathbf{t}P \Leftrightarrow yz \in \mathbf{t}P) : y\}$$

Notice that

$$(\forall x, y, a :: x \in [y]_P \Rightarrow xa \in [ya]_P)$$

It follows that occurrence of an event in some state brings the system in one (new) unique equivalence class. Each equivalence class is a state. State graphs are deterministic, i.e., from each state the occurrence of an event leads to exactly one (other) state. If for some state $[x]_P$ we have that $x \in \mathbf{b}P$ then $[x]_P \in B$ and similar for $\mathbf{t}P$ and T. The initial state equals $[\epsilon]_P$. It can be shown that

$$(\forall x : x \in (\mathbf{a}P)^* : [x]_P \subseteq \mathbf{b}P \ \vee \ [x]_P \cap \mathbf{b}P = \emptyset)$$

and similar for $\mathbf{t}P$.

δ can now be constructed for $x \in (\mathbf{a}P)^*$ and $a \in \mathbf{a}P$ as

$$\delta([x]_P, a) = [xa]_P$$

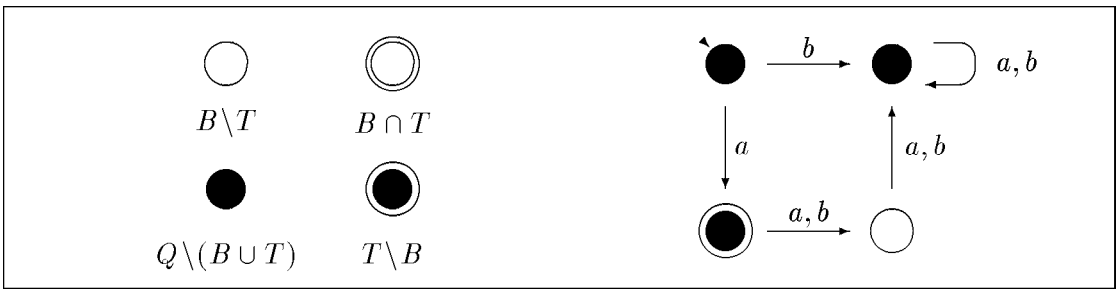
Summerized we have:

Definition 2.2 *For some DES P the state graph $\mathbf{sg}(P) = (\mathbf{a}P, Q, \delta, q, B, T)$ is the corresponding representation, where*

$$\begin{array}{lcl} Q & = & \{x : x \in (\mathbf{a}P)^* : [x]_P\} \\ q & = & [\epsilon]_P \\ B & = & \{x : x \in \mathbf{b}P : [x]_P\} \\ T & = & \{x : x \in \mathbf{t}P : [x]_P\} \\ (\forall x, a : x \in (\mathbf{a}P)^* \ \wedge \ a \in \mathbf{a}P : \delta([x]_P, a) = [xa]_P) & & \end{array}$$

Q, q, B, and T will be denoted by $\mathbf{S}(P)$, $\mathbf{q}(P)$, $\mathbf{B}(P)$, and $\mathbf{T}(P)$ respectively. □

State graphs can be displayed as is shown in figure 2.1 (left). With an open circle we denote a behaviour state and with a solid circle we denote a non-behaviour state. With a larger concentric circle we denote a task state. This leads to the four combinations as are given in the figure.



Figuur 2.1: Left: displaying of different states; Right: system from example 2.3

Example 2.3 Consider the system

$$P = \langle \{a, b\}, (aa|ab), (a) \rangle$$

The following equivalence classes can be found:

$$\begin{array}{lllll} p_0 & = & [\epsilon] & = & \{\epsilon\} \\ p_1 & = & [a] & = & \{a\} \\ p_2 & = & [aa] & = & \{aa, ab\} \\ p_3 & = & [b] & = & \{a, b\}^* \backslash (p_0 \cup p_1 \cup p_2) \end{array}$$

In figure 2.1 the corresponding graph is shown. Notice that $\mathbf{B}(P) = \{p_2\}$, $\mathbf{T}(P) = \{p_1\}$, $\mathbf{q}(P) = p_0$, and $\mathbf{S}(P) = \{p_0, p_1, p_2, p_3\}$. In the diagram the initial state is denoted by an additional small arrow. □

2.1.2 From state graph to DES

Given some state graph G the corresponding DES equals

$$\langle A, \mathbf{b}P, \mathbf{t}P \rangle$$

with

$$\mathbf{b}P = \{x : x \in A^* \ \wedge \ \delta^*(q, x) \in B : x\}$$
$$\mathbf{t}P = \{x : x \in A^* \ \wedge \ \delta^*(q, x) \in T : x\}$$

With $\mathbf{des}(G)$ we denote the DES represented by the graph G.

Example 2.4 Consider the graph given in figure 2.2. It is easily seen that the following DES is represented by that graph:

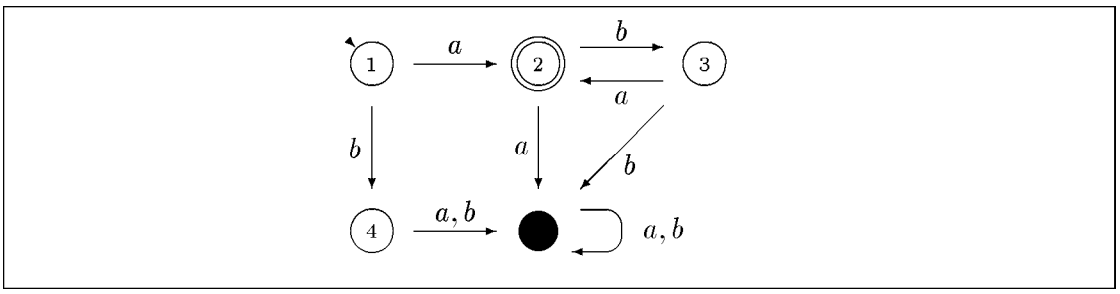
$$\langle \{a, b\}, \mathbf{pref}(b|a(ba)^*), a(ba)^* \rangle$$

□

Property 2.5

$$\mathbf{des}(\mathbf{sg}(P)) = P$$

□



Figuur 2.2: State graph for system P from example 2.4

In general, we do not have this the other way round, i.e.,

$$\mathbf{sg}(\mathbf{des}(G)) \neq G$$

because more state graphs exist that correspond to the same DES.

Three special graphs are

$$\begin{aligned} G_{\text{empty}}(A) &= (A, \{q\}, \delta, q, \emptyset, \emptyset) \\ G_{\text{chaos}}(A) &= (A, \{q\}, \delta, q, \{q\}, \{q\}) \end{aligned}$$

with $\delta(q, a) = q$ for all $a \in A$, and

$$G_{\text{skip}}(A) = (A, \{q, p\}, \delta, q, \{p\}, \{p\})$$

with $\delta(q, a) = p$ and $\delta(p, a) = p$ for all $a \in A$.

Property 2.6

$$\begin{aligned} &\mathbf{des}(G_{\text{empty}}(A)) = \mathbf{empty}(A) \\ &\mathbf{des}(G_{\text{skip}}(A)) = \mathbf{skip}(A) \\ &\mathbf{des}(G_{\text{chaos}}(A)) = \mathbf{chaos}(A) \end{aligned}$$

□

2.1.3 Reachable subgraph

Because we deal with paths in a graph starting in the initial state it has no effect if states ared added that cannot be reached from the initial state. Such states can, if they exists, also easily be eliminated.

Definition 2.7 *Given a state graph G, its* reachable subgraph *is defined by*

$$\mathbf{reachable}(G) = (A, \overline{Q}, \delta|_{\overline{Q}}, q, \overline{B}, \overline{T})$$

where

$$\begin{aligned} \overline{Q} &= \{x : x \in A^* : \delta^*(q, x)\} \\ \overline{B} &= B \cap \overline{Q} \\ \overline{T} &= T \cap \overline{Q} \end{aligned}$$

□

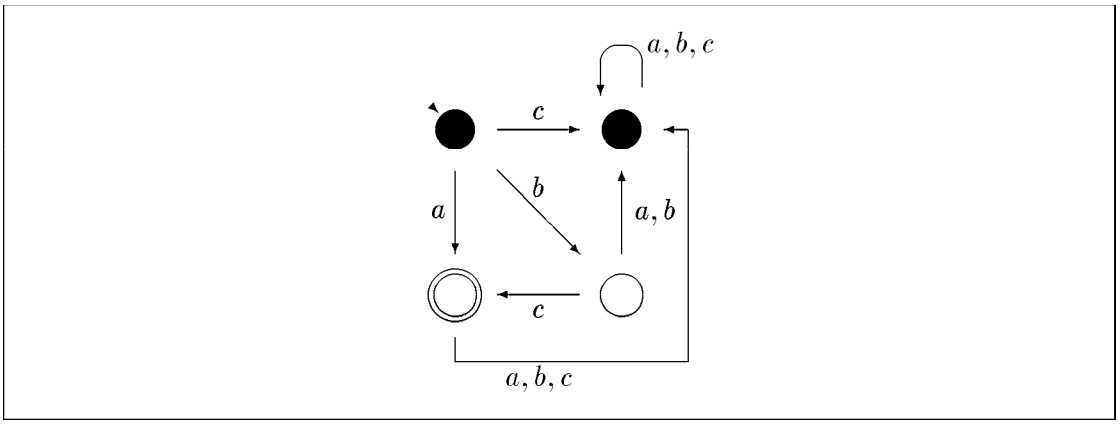
The notation $\delta|_{\overline{Q}}$ stands for δ restricted to $\overline{Q}$. Notice that $\delta|_{\overline{Q}}(p, a) = \delta(p, a) \in \overline{Q}$ for all $q \in \overline{Q}$ and $a \in A$.

Property 2.8

$$\mathbf{des}(\mathbf{reachable}(\mathbf{sg}(P))) = P$$

□

Exercise 2.1 Give a state graph representation for $P = \langle \{a, b, c\}, (a|ab|c^*ac^*|bc^*) \rangle$. Compute $\mathbf{des}(G)$ for G given in figure 2.3. □



Figuur 2.3: State graph for exercise 2.1

2.1.4 Modelling using state graphs

Reconsider the puzzle example from chapter 1. The following events can be thought of:

- w farmer takes wolf to the other side
- g farmer takes goat to the other side
- c farmer takes cabbage to the other side
- f farmer goes to the other side alone
- e one item is eaten

The configuration can be in 17 different behaviour states: each of the items (including the farmer) can be on this side of the river or on the other side, resulting in $4 \times 4 = 16$ states; and one item can be eaten (the resulting configuration is then of no importance to us any more), resulting in one extra state. We can encode the states using a binary number $ijkl$ with:

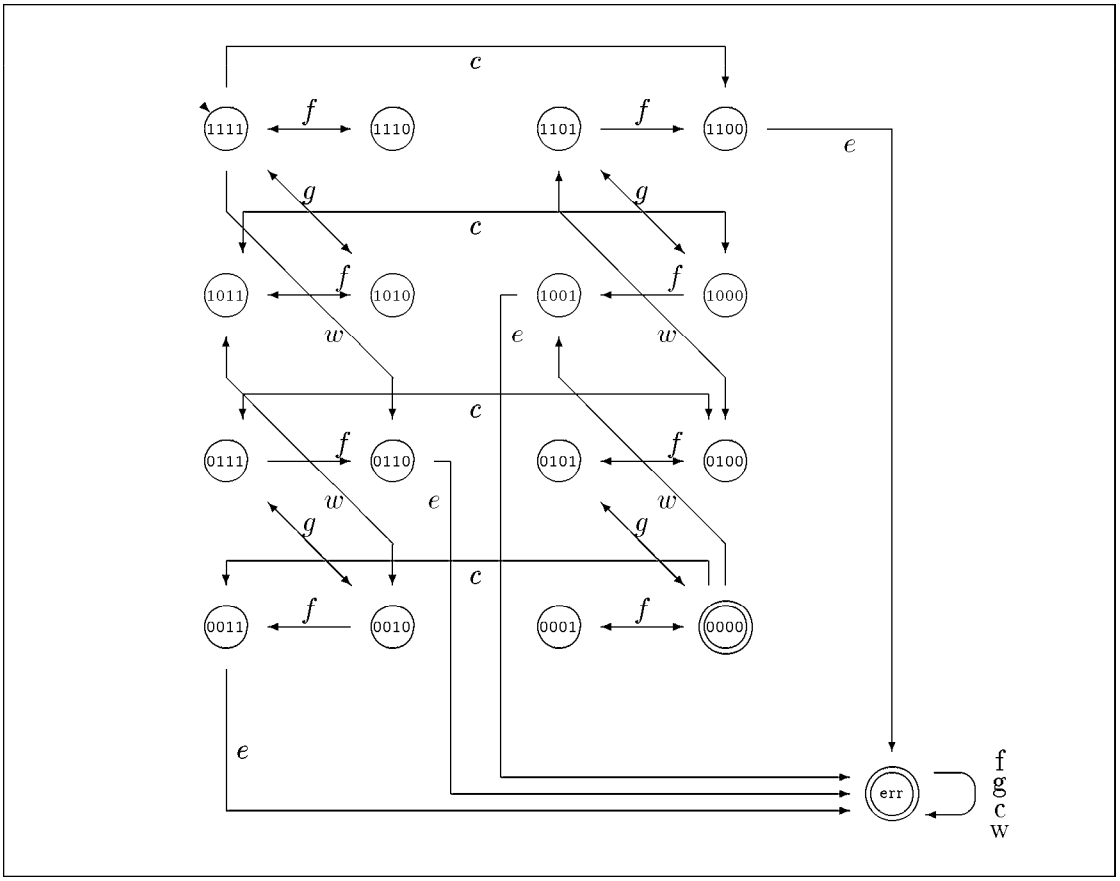
$i = 1$	wolf is here	$i = 0$	wolf is there
$j = 1$	goat is here	$j = 0$	goat is there
$k = 1$	cabbage is here	$k = 0$	cabbage is there
$l = 1$	farmer is here	$l = 0$	farmer is there

Moreover we name the state in which an item is eaten by "err." The state transitions can now easily be found (x denotes 0 or 1):

$$\left.\begin{array}{lcll} xxx1 & \overset{\text{f}}{\longleftrightarrow} & xxx0 & 1100 \\ 1xx1 & \overset{\text{w}}{\longleftrightarrow} & 0xx0 & 1001 \\ x1x1 & \overset{\text{g}}{\longleftrightarrow} & x0x0 & 0110 \\ xx11 & \overset{\text{c}}{\longleftrightarrow} & xx00 & 0011 \end{array}\right\} \xrightarrow{\text{e}} \text{err}$$

Each transition that is not mentioned above is a transition to a non-behaviour/non-task state. Once in this state each transition will return to that state. This state is omitted in the diagram.[1] We start in the state with all items at this side , i.e., state 1111. The only task state is 0000 (all items at the other side). Dependent on the way of modelling

[1] In drawings of state graphs that contain one such a state, sometimes called an dump-state and only needed to get a complete transition function, this state is often omitted as well as all transitions going to it.



Figuur 2.4: State graph representation for the farmer-problem. Dump states are omitted in the graph.

you can also consider state "err" to be a task state (then also the task of the farmer is completed, although not successfully). We find the graph as given in figure 2.4.

This example shows that state graphs can be of great help in modelling some system. It is much more difficult in this case to give the behaviour and task set immediately.

2.2 Regular systems and minimal graphs

If a state graph representation of a discrete system contains a finite number of states we say the system is *regular*.

As said before, more state graphs exists, that represent a DES P. However, the one found using the construction with equivalence classes has a minimum number of states. Therefore, we say $\mathbf{sg}(P)$ is the minimal representation for P.

Property 2.9 $\mathbf{sg}(P)$ *is a minimal representation for* P

proof: We only give an outline of the proof. Suppose $G = (\mathbf{a}P, Q, \delta, q, B, T)$ is another state graph for P. Denote with $\sigma(p)$ all paths starting in q that end in $p \in Q$, i.e.,

$$\sigma(p) = \{x : x \in (\mathbf{a}P)^* \ \wedge \ \delta^*(q, x) = p : x\}$$

For each path in $\sigma(p)$ the future is the same, i.e.,

$$\begin{aligned}(\forall y_1, y_2 : y_1 \in \sigma(p) \ \wedge \ y_2 \in \sigma(p) \\ : (\forall z : z \in (\mathbf{a}P)^* : y_1 z \in \mathbf{b}P \Leftrightarrow y_2 z \in \mathbf{b}P) \\ : (\forall z : z \in (\mathbf{a}P)^* : y_1 z \in \mathbf{t}P \Leftrightarrow y_2 z \in \mathbf{t}P))\end{aligned}$$

which means that

$$\sigma(p) \subseteq [x]_P \quad \text{with } y \in \sigma(p)$$

All equivalence classes of P together just fill up $(\mathbf{a}P)^*$, from which we can conclude that there should be at least as many states p as there are equivalence classes. Therefore, $\mathbf{sg}(P)$ is a minimal representation. □

For each arbitrary state graph G an equivalent minimal state graph can be found, namely $\mathbf{sg}(\mathbf{des}(G))$. However, because we use two kinds of final states in one graph, standard techniques (see [HU79] or [Sne85]) to minimize a finite automaton can not be used. We should use some extension of the standard technique.

2.3 Nondeterministic graphs

State graphs as described above have the disadvantage of being large (in general). Sometimes we can reduce the size of a state graph by considering nondeterministic state graphs. Below we give a definition of a nondeterministic state graph and show that each system can as well be displayed using this kind of graphs.

Definition 2.10 *A* nondeterministic state graph *(nd-graph)* G_{nd} *is defined by*

$$G_{\text{nd}} = (A, Q, \gamma, q, B, F)_{\text{nd}}$$

with A, Q, q, B, and T as in definition 2.2 and

$$\gamma : Q \times (A \cup \{\epsilon\}) \to \mathbf{2}^Q$$

the state transition map. □

Again γ is supposed to be a total function, possibly mapping tuples $(p, a) \in Q \times (A \cup \{\epsilon\})$ onto $\emptyset$.

An nd-graph may have arrows from one state to another labelled with ϵ and also more than one arrow starting in some state with the same label. Therefore, the state transition map γ denotes here a relation on Q, with $q \in \gamma(p, a)$ if there is some transition labelled a from p to q.

Again, we need the closure of the transition γ.

Definition 2.11 *The* closure *of* γ*, denoted by* γ^**, is defined by*

$$\begin{aligned}\gamma^*(p, \epsilon) &= \{p_0, \ldots, p_n : p_0 = p \ \wedge \ (\forall i : 1 \leq i \leq n : p_i \in \gamma(p_{i-1}, \epsilon)) : p_n\} \\ \gamma^*(p, \epsilon) &= \gamma(p, \epsilon) \cup \gamma^*(\gamma(p, \epsilon) \backslash \{p\}, \epsilon) \\ \gamma^*(p, a) &= \gamma^*(\gamma(\gamma^*(p, \epsilon), a), \epsilon) \\ \gamma^*(p, xa) &= \gamma^*(\gamma^*(p, x), a) \\ \gamma^*(\overline{Q}, a) &= \bigcup_{p \in \overline{Q}} \gamma^*(p, a) \qquad \text{for } \overline{Q} \subseteq Q\end{aligned}$$

$\gamma^*(\overline{Q}, x)$ is the set of all states, reachable form a state in $\overline{Q} \subseteq Q$ via a path x including ϵ-transitions. □

An nd-graph also represents a discrete system P, namely

$$\begin{aligned} \mathbf{a}P &= A \\ \mathbf{b}P &= \{x : x \in A^* \ \wedge \ \gamma^*(q,x) \cap B \neq \emptyset : x\} \\ \mathbf{t}P &= \{x : x \in A^* \ \wedge \ \gamma^*(q,x) \cap T \neq \emptyset : x\} \end{aligned}$$

We will use the notation $\mathbf{des}(G_{\mathrm{nd}})$ also for the DES represented by G_{nd}.

Example 2.12 In figure 2.5 (left) (see page 31) an nd-graph is given for

$$P = \langle \{a,b\}, \mathbf{pref}(b|ab), (a|b|ab) \rangle$$

□

Given some nd-graph $G_{\mathrm{nd}} = (A, Q, \gamma, q, B, T)_{\mathrm{nd}}$ we can construct an equivalent deterministic graph as follows:

Definition 2.13

$$\mathbf{det}(G_{\mathrm{nd}}) = (A, \mathbf{2}^Q, \delta, \overline{q}, \overline{B}, \overline{T})$$

where (for $r \in \mathbf{2}^Q$):

$$\begin{aligned} \delta(r,a) &= \bigcup_{p \in r} \gamma^*(p,a) \\ \overline{q} &= \gamma^*(q,\epsilon) \\ \overline{B} &= \{r : r \cap B \neq \emptyset : r\} \\ \overline{T} &= \{r : r \cap T \neq \emptyset : r\} \end{aligned}$$

□

Notice that each state in $\mathbf{det}(G_{\mathrm{nd}})$ is a set of states of G_{nd}.

The above construction is a generalization of the well-known construction to find the deterministic equivalent of a nondeterministic automaton. Apart from unreachable states, each state in $\mathbf{det}(G_{\mathrm{nd}})$ is the set of states that can be reached from another set by doing zero or more ϵ-moves, followed by one normal move, followed by zero or more ϵ-moves.

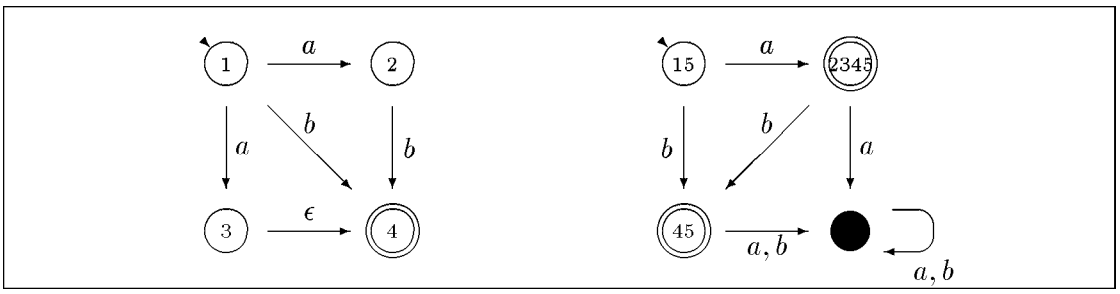
Notice that the above construction also holds for graphs with infinitely many states.

Property 2.14

$$\mathbf{des}(\mathbf{det}(G_{\mathrm{nd}})) = \mathbf{des}(G_{\mathrm{nd}})$$

proof: The alphabets are the same. Let $G_{\mathrm{nd}} = (A, Q, \gamma, q, B, T)_{\mathrm{nd}}$. For the behaviour sets we have:

$$\begin{aligned} & \mathbf{b}(\mathbf{des}(\mathbf{det}(G_{\mathrm{nd}}))) \\ = \quad & [\text{ definition of } \mathbf{des} \text{ }] \\ & \{x : \delta^*(\overline{q}, x) \in \overline{B} : x\} \\ = \quad & [\text{ definition of } \delta \text{ in } \mathbf{det} \text{ }] \\ & \{x : \bigcup_{p \in \overline{q}} \gamma^*(p,x) \in \overline{B} : x\} \\ = \quad & [\text{ shorthand notation for } \gamma^* \text{ }] \\ & \{x : \gamma^*(\overline{q}, x) \in \overline{B} : x\} \\ = \quad & [\text{ definition of } \overline{q} \text{ in } \mathbf{det} \text{ }] \end{aligned}$$



Figuur 2.5: Left: An nd-graph representing the system from example 2.12: Right: Deterministic graph for this system

$$
\begin{aligned}
& \{x : \gamma^*(\gamma^*(q,\epsilon),x) \in \overline{B} : x\} \\
= \quad & [\text{ definition of } \gamma^* \text{ }] \\
& \{x : \gamma^*(q,x) \in \overline{B} : x\} \\
= \quad & [\text{ definition of } \overline{B} \text{ in } \mathbf{det} \text{ }] \\
& \{x : \gamma^*(q,x) \cap B \neq \emptyset : x\} \\
= \quad & [\text{ definition of } \mathbf{des}(G_{\mathrm{nd}}) \text{ }] \\
& \mathbf{b}(\mathbf{des}(G_{\mathrm{nd}}))
\end{aligned}
$$

Similar for **t**. □

Example 2.15 (continuation of example 2.12)
We can use the above construction on the graph of figure 2.5 (left) to get a deterministic graph for the same system. Let $Q = \{p_1, p_2, p_3, p_4, p_5\}$ be the set of states of this graph. Then we have:

$$
\begin{array}{lcll}
\gamma^*(q,\epsilon) & = & \{p_1\} & = \quad \overline{q} \\
\gamma^*(\{p_1\},a) & = & \{p_2,p_3,p_4\} & \\
\gamma^*(\{p_1\},b) & = & \{p_4\} & \\
\gamma^*(\{p_2,p_3,p_4\},a) & = & \emptyset & \\
\gamma^*(\{p_2,p_3,p_4\},b) & = & \{p_4\} & \\
\gamma^*(\{p_4\},a) & = & \emptyset & \\
\gamma^*(\{p_4\},b) & = & \emptyset & \\
\gamma^*(\emptyset,a) & = & \emptyset & \\
\gamma^*(\emptyset,b) & = & \emptyset &
\end{array}
$$

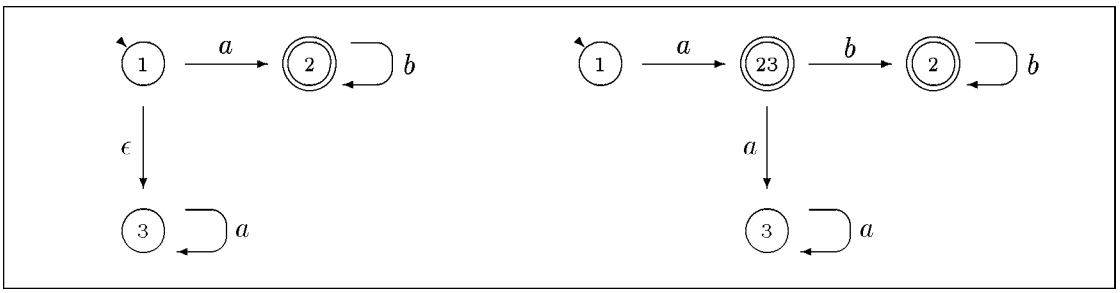
which leads to the graph of figure 2.5 (right). Notice that we only have examined the reachable states, i.e., only considered a subset of $\mathbf{2}^Q$. The behaviour and task states are

$$
\begin{array}{lcl}
\overline{B} & = & \{\{p_1\},\{p_2,p_3,p_4\},\{p_4\}\} \\
\overline{T} & = & \{\{p_2,p_3,p_4\},\{p_4\}\}
\end{array}
$$

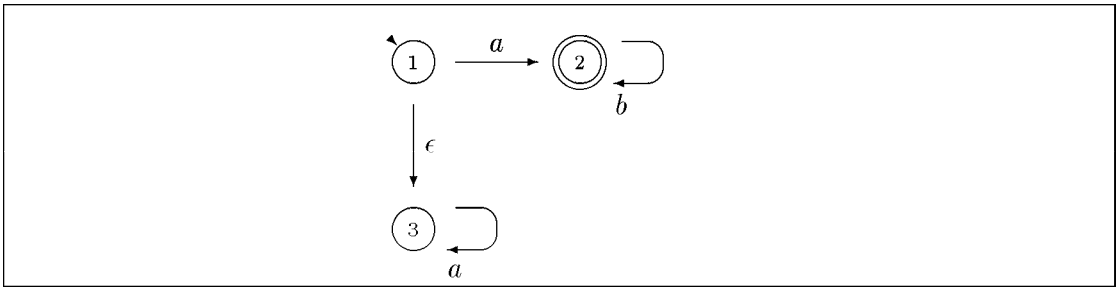
The state $\{\emptyset\}$ is the only non-behaviour state. It can easily be checked that this graph also represents the system

$$P = \langle \{a,b\}, \mathbf{pref}(b|ab), (a|b|ab)\rangle$$

□



Figuur 2.6: nd-graph for remark 2.16



Figuur 2.7: State graph for exercise 2.2

Remark 2.16 The ϵ-transitions in an nd-graph can also be interpreted as so called silent moves. In that case the behaviour of system P as displayed using the nd-graph of figure 2.6 should be explained as: occurrence of a and repeated occurrence of b or a silent move and then only occurrences of a. If the silent move occurs before the first a, then only repeated a's are possible, if event a occurs first, then only behaviour ab^* is possible.

Milner, see [Mil80], investigated a specific operational interpretation of silent moves. We do not use this interpretation. In fact, we consider left and right graph in figure 2.6 to be equivalent, i.e., both represent the same system. We use nd-graphs only as a way to display a system. We do not use the graph for interpretations. So we use the angelic interpretation of nondeterminism (as is usual in automata theory) and not the demonic interpretation of [Mil80] (See [Hes90]). □

Exercise 2.2 Give $\mathbf{des}(G_{\mathrm{nd}})$ for G_{nd} as given in figure 2.7. Compute also $\mathbf{det}((G_{\mathrm{nd}})$ and check that $\mathbf{des}(G_{\mathrm{nd}}) = \mathbf{des}(\mathbf{det}((G_{\mathrm{nd}}))$. □

2.4 Alphabet restriction in state graphs

Alphabet restriction in state graphs can be done by replacing every label in the graph that is not in the alphabet by ϵ. We find a non-deterministic graph that represents the system after alphabet restriction.

Definition 2.17 *Associated with a state graph G and an alphabet B we define the graph G restricted to B, denoted by $G\lceil B$, by*

$$G\lceil B = \mathbf{det}(G_{\mathrm{nd}}) \qquad \textit{with: } G_{\mathrm{nd}} = (A \cap B, Q, \gamma, q, B, T)_{\mathrm{nd}}$$

with

$$\begin{aligned}\gamma(p,\epsilon) &= \bigcup_{a\in A\setminus B} \delta(p,a)\\ \gamma(p,a) &= \{\delta(p,a)\} \qquad \textit{for } a\in A\cap B\end{aligned}$$

□

Property 2.18

$$\mathbf{des}(\mathbf{sg}(P)\lceil A) = P\lceil A$$

proof: Suppose $\mathbf{sg}(P) = (\mathbf{a}P, Q, \delta, q, B, T)$, then

$$\begin{aligned}
&\mathbf{b}(\mathbf{des}(\mathbf{sg}(P)\lceil A))\\
=\quad &\text{[definition 2.17]}\\
&\mathbf{b}(\mathbf{des}(\mathbf{det}(\mathbf{a}P\cap A, Q, \gamma, q, B, T)_{\mathrm{nd}}))\\
=\quad &\text{[property 2.14]}\\
&\mathbf{b}(\mathbf{des}(\mathbf{a}P\cap A, Q, \gamma, q, B, T)_{\mathrm{nd}})\\
=\quad &\text{[construction of } \mathbf{des} \text{ for nd-graphs]}\\
&\{y : y\in(\mathbf{a}P\cap A)^* \ \wedge\ \gamma^*(q,y)\cap B\neq\emptyset : y\}\\
=\quad &\text{[see below]}\\
&\{y : (\exists x : x\in(\mathbf{a}P)^* \ \wedge\ \delta^*(q,x)\in B : x\lceil A = y) : y\}\\
=\quad &\\
&\{x : x\in(\mathbf{a}P)^* \ \wedge\ \delta^*(q,x)\in B : x\lceil A\}\\
=\quad &\text{[construction of } \mathbf{sg}(P) \text{]}\\
&\{x : x\in(\mathbf{a}P)^* \ \wedge\ x\in\mathbf{b}P : x\lceil A\}\\
=\quad &\\
&\mathbf{b}P\lceil A
\end{aligned}$$

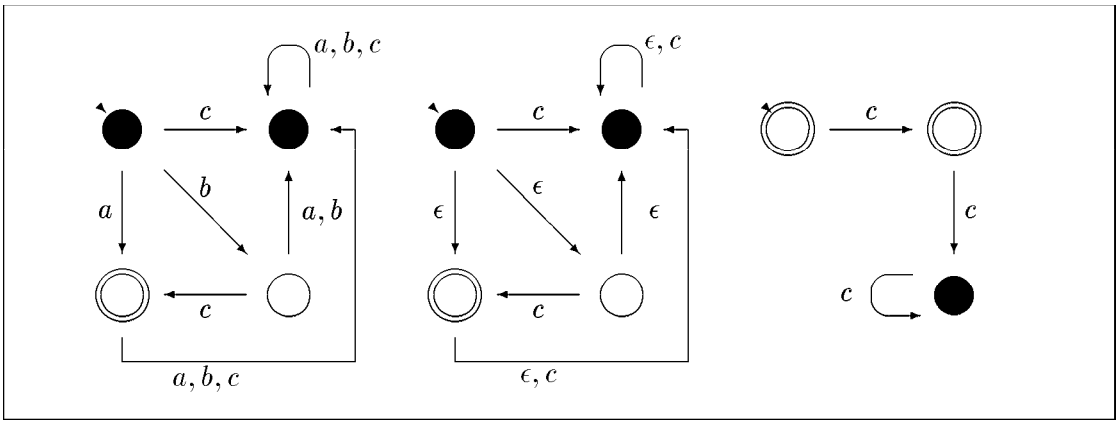
where we have used the fact that:

$$\gamma^*(q,y)\cap B\neq\emptyset \quad\Leftrightarrow\quad (\exists x : x\in(\mathbf{a}P)^* \ \wedge\ \delta^*(q,x)\in B : x\lceil A = y)$$

which can be proved by induction on the length of y. Take $y=\epsilon$, then we have

$$\begin{aligned}
&\gamma^*(q,\epsilon)\cap B\neq\emptyset\\
\Leftrightarrow\quad &\text{[definition of } \gamma^*(q,c) \text{]}\\
&\{p_0,\ldots,p_n : p_0 = q \ \wedge\ (\forall i : 1\le i\le n : p_i\in\gamma(p_{i-1},\epsilon)) : p_n\}\cap B\neq\emptyset\\
\Leftrightarrow\quad &\text{[definition 2.17]}\\
&\{p_0,\ldots,p_n : p_0 = q \ \wedge\ (\forall i : 1\le i\le n : p_i\in \bigcup_{a\in\mathbf{a}P\setminus A}\delta(p_{i-1},a) : p_n\}\cap B\neq\emptyset\\
\Leftrightarrow\quad &\\
&\{p_0,\ldots,p_n,a_1,\ldots,a_n : p_0 = q \wedge\\
&\quad (\forall i : 1\le i\le n : a_i\in\mathbf{a}P\setminus A \ \wedge\ p_i = \delta(p_{i-1},a_i))) : p_n\}\cap B\neq\emptyset\\
\Leftrightarrow\quad &\\
&(\exists a_1,\ldots a_n :: (\forall i :: 1\le i\le n : a_i\in\mathbf{a}P\setminus A \ \wedge\ \delta^*(q,a_1a_2\cdots a_n)\in B))\\
\Leftrightarrow\quad &\text{[take } x = a_1a_1\cdots a_n \text{]}\\
&(\exists x : x\in(\mathbf{a}P)^* \ \wedge\ \delta(q,x)\in B : x\lceil A = \epsilon\}
\end{aligned}$$

The induction step is similar. □



Figuur 2.8: $\mathbf{sg}(P)$ *(left), its restriction to* $\{c\}$ *(middle), and its deterministic equivalence (right)*

Example 2.19 Consider the system

$$P = \langle \{a, b, c\}, \{b, bc, a\}, \{a, bc\} \rangle$$

its corresponding graph $\mathbf{sg}(P)$ is given in figure 2.8. Also in that figure, the graph $\mathbf{sg}(P)\lceil\{c\}$ is given constructed using definition 2.17. Making this graph deterministic leads to a graph representing the system

$$\langle \{c\}, \{\epsilon, c\}, \{\epsilon, c\} \rangle$$

which is equal to $P\lceil\{c\}$. □

Exercise 2.3 Construct $\mathbf{sg}(\langle \{a, b, c\}, (a|b|bc), bc)$. Construct the graph for $P\lceil\{a, b\}$ and its deterministic equivalent. □

2.5 State graph for a connection

It is straightforward to construct a graph for the connection of two systems P and R: just take the cartesian product of the two state sets. Transitions go from state (i, j) to state (i', j) labelled a if event a is only an event of the first system and that system has a transition from state i to state i' labelled with a. Similar for (i, j) to (i, j') if a is only an event of the second system and that system has a transition from state j to j' labelled with a. If an event a is common to both systems that event can only occur in the connection if both systems can engage in it. Therefore, a transition from (i, j) to (i', j') labelled with such a common a is possible only if the first system has a transition from i to i' labelled with a and the second system has a transition form j to j' labelled with a. This leads to the following definition:

Definition 2.20 *Given two state graphs* $G_i = (A_i, Q_i, \delta_i, q_i, B_i, T_i)$ $(i = 1, 2)$*, we define the* connection graph *by*

$$G_1 \parallel G_2 = (A_1 \cup A_2, Q_1 \times Q_2, \delta, (q_1, q_2), B_1 \times B_2, T_1 \times T_2)$$

where

$$\begin{array}{rcll} \delta((p_1,p_2),a) & = & (\delta_1(p_1,a),p_2) & \text{if } a \in A_1 \setminus A_2 \\ & = & (p_1,\delta_2(p_2,a)) & \text{if } a \in A_2 \setminus A_1 \\ & = & (\delta_1(p_1,a),\delta(p_2,a)) & \text{if } a \in A_1 \cap A_2 \end{array}$$ □

In order to prove that this construction leads to a state graph for the connection, we need the following property:

Property 2.21 *For two state graphs G_i as in definition 2.20 we have:*

$(a)\quad \delta^*((q_1,q_2),x) \in B_1 \times B_2 \quad \Leftrightarrow \quad \delta_1^*(q_1,x\lceil A_1) \in B_1 \ \wedge\ \delta_2^*(q_2,x\lceil A_2) \in B_2$
$(b)\quad \delta^*((q_1,q_2),x) \in T_1 \times T_2 \quad \Leftrightarrow \quad \delta_1^*(q_1,x\lceil A_1) \in T_1 \ \wedge\ \delta_2^*(q_2,x\lceil A_2) \in T_2$

proof: The proof is by induction on the length of x:
First, suppose $x = \epsilon$, then

$$\begin{array}{ll} & \delta^*((q_1,q_2),\epsilon) \in B_1 \times B_2 \\ \Leftrightarrow & \quad [\text{ definition of } \delta^*] \\ & (q_1,q_2) \in B_1 \times B_2 \\ \Leftrightarrow & \\ & q_1 \in B_1 \ \wedge\ q_2 \in B_2 \\ \Leftrightarrow & \quad [\text{ definition of } \delta^*] \\ & \delta_1^*(q_1,\epsilon) \in B_1 \ \wedge\ \delta_2^*(q_2,\epsilon) \in B_2 \end{array}$$

Next, suppose (a) holds for some x, then for any a we have

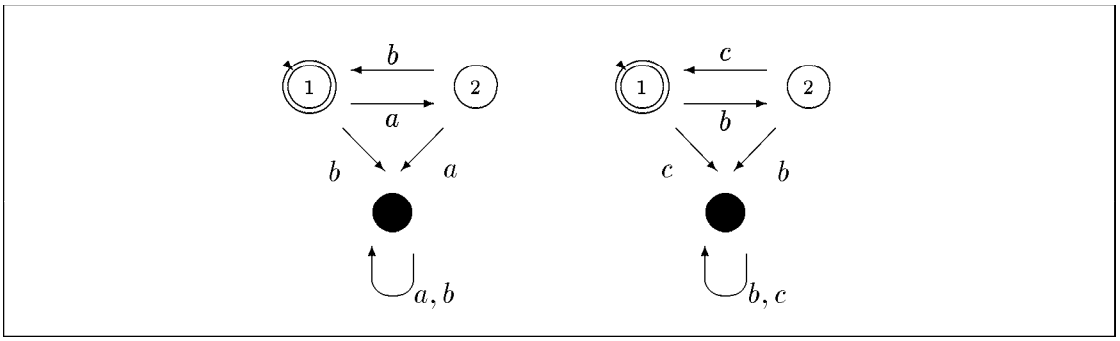
$$\begin{array}{ll} & \delta^*((q_1,q_2),xa) \in B_1 \times B_2 \\ \Leftrightarrow & \quad [\text{ definition of } \delta^*] \\ & \delta(\delta^*(q_1,q_2),x),a) \in B_1 \times B_2 \\ \Leftrightarrow & \quad [\text{ assumption }] \\ & p_1 = \delta_1^*(q_1,x\lceil A_1) \in B_1 \ \wedge\ p_2 = \delta_2^*(q_2,x\lceil A_2) \in B_2 \ \wedge\ \delta((p_1,p_2),a) \in B_1 \times B_2 \\ \Leftrightarrow & \quad [\text{ definition 2.20 }] \\ & p_1 = \delta_1^*(q_1,x\lceil A_1) \in B_1 \ \wedge\ p_2 = \delta_1^*(q_2,x\lceil A_2) \in B_2 \\ & \quad \wedge \begin{cases} \delta_1(p_1,a) \in B_1 & \text{if } a \in A_1 \setminus A_2 \\ \delta_2(p_2,a) \in B_2 & \text{if } a \in A_2 \setminus A_1 \\ \delta_1(p_1,a) \in B_1 \ \wedge\ \delta_2(p_2,a) \in B_2 & \text{if } a \in A_1 \cap A_2 \end{cases} \\ \Leftrightarrow & \quad [\text{ shorter notation }] \\ & p_1 = \delta_1^*(q_1,x\lceil A_1) \in B_1 \ \wedge\ \delta_1^*(p_1,a\lceil A_1) \in B_1 \ \wedge \\ & p_2 = \delta_1^*(q_2,x\lceil A_2) \in B_2 \ \wedge\ \delta_2^*(p_2,a\lceil A_2) \in B_2 \\ \Leftrightarrow & \quad [\text{ definition of } \delta^*] \\ & \delta_1^*(q_1,(xa)\lceil A_1) \in B_1 \ \wedge\ \delta_2^*(q_2,(xa)\lceil A_2) \in B_2 \end{array}$$

The proof of (b) is similar. □

Property 2.22

$$\mathbf{des}(\mathbf{sg}(P_1) \parallel \mathbf{sg}(P_2)) = P_1 \parallel P_2$$

proof: The alphabets of lhs. and rhs. are identical. Let $B_i = \mathbf{B}(P_i)$ and $q_i = \mathbf{q}(P_i)$. For the behaviour sets we have:



Figuur 2.9: State graphs G_1 (left) and G_2 (right) for example 2.23

$\mathbf{b}(\mathbf{des}(\mathbf{sg}(P_1) \parallel \mathbf{sg}(P_2)))$
= [definition 2.20 and definition of **des**]
$\{x : x \in (\mathbf{a}P_1 \cup \mathbf{a}P_2)^* \ \wedge \ \delta^*((q_1, q_2), x) \in B_1 \times B_2 : x\}$
= [property 2.21]
$\{x : x \in (\mathbf{a}P_1 \cup \mathbf{a}P_2)^* \ \wedge \ \delta_1^*(q_1, x\lceil \mathbf{a}P_1) \in B_1 \ \wedge \ \delta_2^*(q_2, x\lceil \mathbf{a}P_2) \in B_2 : x\}$
= [construction of $\mathbf{sg}(P_i)$]
$\{x : x \in (\mathbf{a}P_1 \cup \mathbf{a}P_2)^* \ \wedge \ x\lceil \mathbf{a}P_1 \in \mathbf{b}P_1 \ \wedge \ x\lceil \mathbf{a}P_2 \in \mathbf{b}P_2 : x\}$
= [definition of $\parallel$]
$x \in \mathbf{b}(P_1 \parallel P_2)$

Similar for **t**. □

Example 2.23 Consider the graphs as given in figure 2.9. According to the previous property the graph for the interaction is as given in figure 2.10. Notice that this graph is not minimal: all non-behaviour states can be replaced by one non-behaviour state. □

Once alphabet restriction and connection of graphs are defined we can easily find a method to compute the blend of two graphs:

Property 2.24

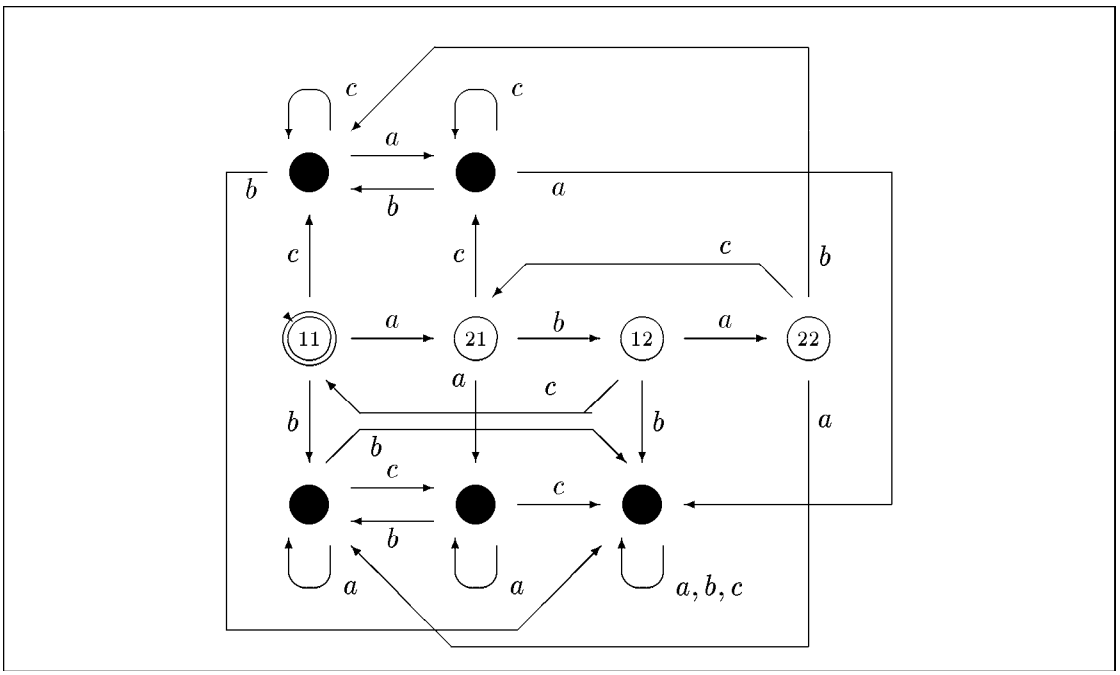
$$\mathbf{des}((\mathbf{sg}(P) \parallel \mathbf{sg}(R))\lceil(\mathbf{a}P \div \mathbf{a}R)) = P \rceil\!\parallel R$$

proof:

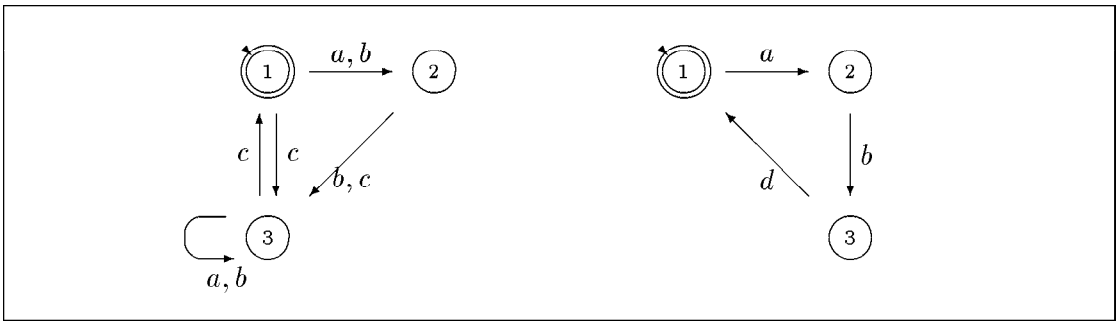
$\mathbf{des}((\mathbf{sg}(P) \parallel \mathbf{sg}(R))\lceil(\mathbf{a}P \div \mathbf{a}R))$
= [property 2.18 with $\mathbf{sg}(S) = \mathbf{sg}(P) \parallel \mathbf{sg}(R)$]
$S\lceil(\mathbf{a}P \div \mathbf{a}R)$
= [$S = \mathbf{des}(\mathbf{sg}(S)) = \mathbf{des}(\mathbf{sg}(P) \parallel \mathbf{sg}(R))$]
$\mathbf{des}(\mathbf{sg}(P) \parallel \mathbf{sg}(R))\lceil(\mathbf{a}P \div \mathbf{a}R)$
= [property 2.22]
$(P \parallel R)\lceil(\mathbf{a}P \div \mathbf{a}R)$
= [definition of $\rceil\!\lceil$]
$P \rceil\!\lceil R$

□

Exercise 2.4 Compute $G_1 \parallel G_2$ and $G_1 \rceil\!\lceil G_2$ and the corresponding minimal graphs for G_1 and G_2 as given in figure 2.11. □



Figuur 2.10: Connection of G_1 and G_2



Figuur 2.11: State graphs G_1 (left) and G_2 (right) for exercise 2.4.

2.6 State graph for a reflection

The reflection performed on state graphs can now be defined by

Definition 2.25 *The* dual graph *of G is defined by*

$$\sim G = (A, Q, \delta, q, Q \setminus B, Q \setminus T)$$

□

$\sim G$ is the complement of G. Each state gets its complementary meaning: a behaviour state becomes a non-behaviour state, etc.

Property 2.26

$$\mathbf{des}(\sim \mathbf{sg}(P)) = \sim P$$

proof: Let $\mathbf{sg}(P) = (A, Q, \delta, q, B, T)$, then:

$$
\begin{array}{ll}
 & \mathbf{b}(\mathbf{des}(\sim\mathbf{sg}(P))) \\
= & \quad [\text{ definition of } \mathbf{des} \text{ and definition 2.25 }] \\
 & \{x : x \in A^* \ \wedge\ \delta^*(q,x) \in Q \setminus B : x\} \\
= & \\
 & \{x : x \in A^* \ \wedge\ \delta^*(q,x) \notin B : x\} \\
= & \quad [\text{ construction of } \mathbf{sg}(P) \,] \\
 & \{x : x \in (\mathbf{a}P)^* \ \wedge\ x \notin \mathbf{b}P : x\} \\
= & \\
 & \mathbf{b}(\sim P)
\end{array}
$$

□

2.7 State graph for the realistic interior

Finding the realistic interior means finding all states reachable from q with paths not going through non-behaviour states:

Definition 2.27 *The* realistic interior *of a graph G is defined by*

$$
\begin{array}{lcll}
\mathbf{real}(G) & = & G_{\text{empty}}(A) & \text{if } q \notin B \\
 & = & \mathbf{reachable}(A, B \cup \{\overline{p}\}, \overline{\delta}, q, B, F \cap B) & \text{otherwise}
\end{array}
$$

where $\overline{p}$ is a fresh state ($\notin B$) and

$$
\begin{array}{lcll}
\overline{\delta}(p,a) & = & \delta(p,a) & \text{if } \delta(p,a) \in B \\
 & = & \overline{p} & \text{otherwise} \\
\overline{\delta}(\overline{p},a) & = & \overline{p} & \text{for all } a \in A
\end{array}
$$

□

Property 2.28

$$\mathbf{des}(\mathbf{real}(\mathbf{sg}(P))) = \mathbf{real}(P)$$

proof: First, notice that

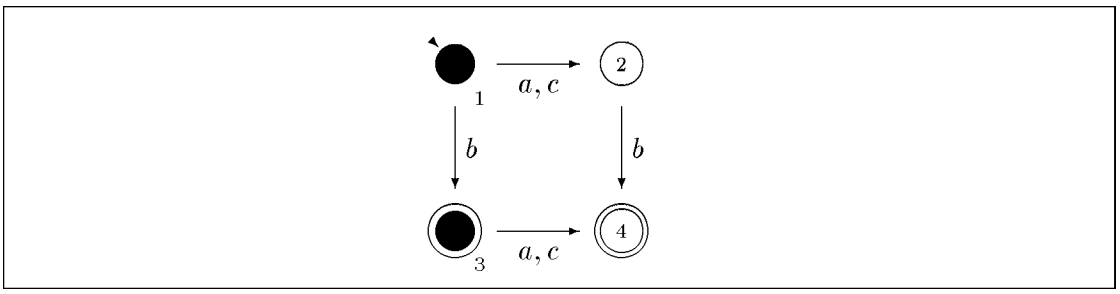
$$
\begin{array}{ll}
 & \overline{\delta}(q, xy) \in B \\
\Rightarrow & \quad [\text{ can easy be proved by induction }] \\
 & \overline{\delta}(q, x) \in B \\
\Rightarrow & \quad [\text{ from the construction }] \\
 & \delta(q, x) \in B
\end{array}
$$

If $q \notin B$ it is evident that the des-interior is empty. So suppose $q \in B$, then we have

$$
\begin{array}{ll}
 & \mathbf{b}(\mathbf{des}(\mathbf{real}(\mathbf{sg}(P)))) \\
= & \quad [\text{ definition of } \mathbf{des} \text{ and definition 2.27 }] \\
 & \{x : x \in A^* \ \wedge\ \overline{\delta}^*(q,x) \in B : x\} \\
= & \quad [\text{ above implication }] \\
 & \{x : x \in A^* \ \wedge\ \delta^*(q,x) \in B \ \wedge\ (\forall y : y \leq x : \delta^*(q,y) \in B) : x\} \\
= & \quad [\text{ construction of } \mathbf{sg}(P) \,] \\
 & \{x : x \in (\mathbf{a}P)^* \ \wedge\ x \in \mathbf{b}P \ \wedge\ (\forall y : y \leq x : y \in \mathbf{b}P) : x\} \\
= & \\
 & \mathbf{b}(\mathbf{real}(P))
\end{array}
$$

□

Exercise 2.5 Compute $\sim G$ with G given in figure 2.12 and its realistic interior. □



Figuur 2.12: State graph G for exercise 2.5

2.8 Other operations on state graphs

Also for union, intersection, and exclusion we can give the corresponding state graphs:

Definition 2.29 *Let* $G_i = (A_i, Q_i, \delta_i, q_i, B_i, T_i)$ *(*$i = 1, 2$*), then we define:*

(1) $G_1 \cup G_2 = (A_1 \cup A_2, Q_1 \times Q_2, \delta, (q_1, q_2), B, T)$
with: $(\forall p_1, p_2, a : p_1 \in Q_1 \ \wedge \ p_2 \in Q_2$
$: \ a \in A_1 \backslash A_2 \ \Rightarrow \ \delta((p_1, p_2), a) = (\delta_1(p_1, a), p_2) \wedge$
$a \in A_2 \backslash A_1 \ \Rightarrow \ \delta((p_1, p_2), a) = (p_1, \delta_2(p_2, a)) \wedge$
$a \in A_1 \cup A_2 \ \Rightarrow \ \delta((p_1, p_2), a) = (\delta_1(p_1, a), \delta_2(p_2, a))$
and: $B = \{p_1, p_2 : p_1 \in B_1 \ \vee \ p_2 \in B_2 : (p_1, p_2)\}$
$T = \{p_1, p_2 : p_1 \in T_1 \ \vee \ p_2 \in T_2 : (p_1, p_2)\}$
(2) $G_1 \cap G2 = (A_1 \cap A_2, Q_1 \times Q_2, \delta, (q_1, q_2), B_1 \times B_2, T_1 \times T_2)$
with: $(\forall p_1, p_2, a : p_1 \in Q_1 \ \wedge \ p_2 \in Q_2 \ \wedge \ a \in A_1 \cap A_2$
$: \delta((p_1, p_2), a) = (\delta_1(p_1, a), \delta_2(p_2, a))$

And if $A_1 = A_2 = A$ *we have:*

(3) $G_1 \backslash G_2 = (A, Q_1 \times Q_2, \delta, (q_1, q_2), B, T)$
with: $(\forall p_1, p_2, a : p_1 \in Q_1 \ \wedge \ p_2 \in Q_2 \ \wedge \ a \in A$
$: \delta((p_1, p_2), a) = (\delta_1(p_1, a), \delta_2(p_2, a))$
and: $B = \{p_1, p_2 : p_1 \in B_1 \ \wedge \ p_2 \notin B_2 : (p_1, p_2)\}$
$T = \{p_1, p_2 : p_1 \in T_1 \ \wedge \ p_2 \notin T_2 : (p_1, p_2)\}$ □

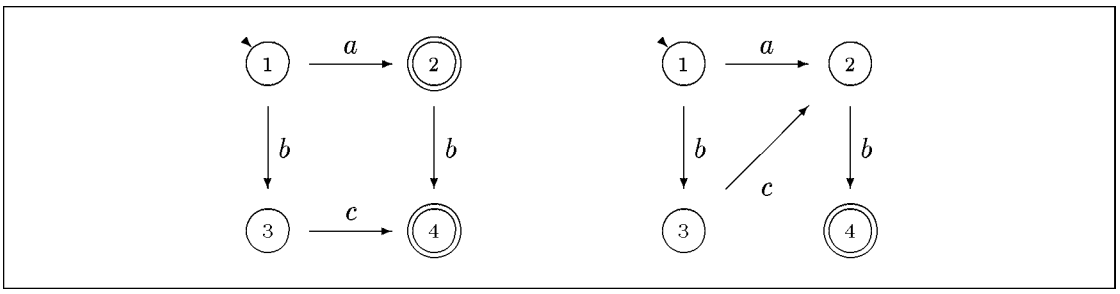
The following property will not come as a surprise:

Property 2.30

(1) $\mathbf{des}(\mathbf{sg}(P) \cup \mathbf{sg}(R)) = P \cup R$
(2) $\mathbf{des}(\mathbf{sg}(P) \cap \mathbf{sg}(R)) = P \cap R$
(3) $\mathbf{des}(\mathbf{sg}(P) \backslash \mathbf{sg}(R)) = P \backslash R$ □

Exercise 2.6 Consider the graphs G_1 and G_2 from figure 2.13, both with alphabet $\{a, b, c\}$. Compute $G_1 \cup G_2$, $G_1 \cap G_2$ and $G_1 \backslash G_2$ and also the minimal state graph representation of each of these results. □

State graphs can be used to determine if some inclusion is valid:



Figuur 2.13: State graphs G_1 and G_2 for exercise 2.6. Both graphs also contain a dump state (with number 5), which is not drawn.

Lemma 2.31

$$\mathbf{reachable}(\mathbf{sg}(P)\backslash\mathbf{sg}(R)) = G_{\text{empty}}(\mathbf{a}P) \;\Rightarrow\; P \subseteq R$$

proof:

$\mathbf{reachable}(\mathbf{sg}(P)\backslash\mathbf{sg}(R)) = G_{\text{empty}}(\mathbf{a}P)$

$\Leftrightarrow$

$\mathbf{des}(\mathbf{reachable}(\mathbf{sg}(P)\backslash\mathbf{sg}(R))) = \mathbf{des}(G_{\text{empty}}(\mathbf{a}P))$

$\Leftrightarrow$ [property 2.8]

$\mathbf{sg}(P)\backslash\mathbf{sg}(R) = G_{\text{empty}}(\mathbf{a}P)$

$\Rightarrow$

$\mathbf{des}(\mathbf{sg}(P)\backslash\mathbf{sg}(R)) = \mathbf{des}(G_{\text{empty}}(\mathbf{a}P))$

$\Leftrightarrow$ [property 2.30 and 2.6]

$P\backslash R = \mathbf{empty}(\mathbf{a}P)$

$\Leftrightarrow$

$\mathbf{t}P \subseteq \mathbf{t}R \;\wedge\; \mathbf{b}P \subseteq \mathbf{b}R$

□

If both $\mathbf{sg}(P)\backslash\mathbf{sg}(R)$ and $\mathbf{sg}(R)\backslash\mathbf{sg}(P)$ result in $G_{\text{empty}}(\mathbf{a}P)$ we have that $\mathbf{sg}(P)$ and $\mathbf{sg}(R)$ represent the same discrete event system. It does not mean that both graphs are the same. They only represent the same systems: it can only be said that they are equivalent.

Representing discrete event systems by state graphs gives the possibility to do a number of computations on computers. The above defined operators on state graphs can easily be programmed. As long as the graphs have a finite number of states (i.e., the systems are regular), computations can be done effectively, i.e., in a finite number of steps.

References

In case of regular discrete event systems state graphs are in fact finite state machines. More about finite state machines can be found in [AU72].

An alternative for state graphs is using recurrent equations as is done in [Hoa85]. The example of the puzzle then becomes:

$$\begin{aligned} \mathit{system} &= f \cdot \mathit{system}_1 | w \cdot \mathit{system}_2 | g \cdot \mathit{system}_3 | \cdots \\ \mathit{system}_1 &= f \cdot \mathit{system} | \ldots \end{aligned}$$

which is, in essence, equivalent with the state graph representation. In fact, each recurrent expression can be identified with a state in the state graph. Each transition in the state graph corresponds to an alternative in the recurrent expression. However, recurrent expressions cannot easily model a DES, because we cannot model holes in the behaviour using such expressions.

Parts of this chapter appeared in [Sme92]. Thanks go to Jan Eppo Jonker for carefully reading earlier manuscripts that lead to this chapter.

3

Locked systems

In this chapter we will investigate more about locked systems, especially situations in which the connection of systems blocks. We study the notions deadlock and livelock and give definitions and methods to get rid of the lock.

3.1 Assumption

In this chapter we will only deal with realistic DESs, i.e., assume that the behaviour is prefix-closed and each task is a behaviour:

$$\begin{array}{lcl} \mathbf{b}P & = & \mathbf{pref}(\mathbf{b}P) \\ \mathbf{t}P & \subseteq & \mathbf{b}P \end{array}$$

3.2 Deadlock

If a system reaches a point in its behaviour in which no event can occur any more, while no task is completed, we speak of deadlock. Formally:

Definition 3.1 *A discrete system P is* deadlocked *in $x \in \mathbf{b}(P)$ if:*

$$x \notin \mathbf{t}P \ \wedge \ (\forall a : a \in \mathbf{a}P : xa \notin \mathbf{b}P)$$

The set of all deadlocked traces is defined by

$$\mathbf{deadlock}(P) = \{x : x \in \mathbf{b}P \ \wedge \ x \notin \mathbf{t}P \ \wedge \ (\forall a : a \in \mathbf{a}P : xa \notin \mathbf{b}P) : x\}$$

A system P is called deadlock free*, notation* $\mathbf{deadlockfree}(P)$*, if*

$$\mathbf{deadlock}(P) = \emptyset$$

□

If $x \in \mathbf{t}P$ we have $x \notin \mathbf{deadlock}(P)$, because completed behaviour can never deadlock (but only finish) the system. A simple example to illustrate deadlock is given below.

Example 3.2 Consider the following systems:

$$\begin{array}{l} P = \langle \{b, c, d\}, (dcb)^* \rangle \\ R = \langle \{b, c, e\}, (ecc)^* \rangle \end{array}$$

Then we find

$$\begin{array}{lcl} \mathbf{t}(P \parallel R) & = & \epsilon \\ \mathbf{b}(P \parallel R) & = & \mathbf{pref}((de|ed)c) \end{array}$$

Take $x = edc$, then $x\lceil \mathbf{a}P \in \mathbf{b}P$ and $x\lceil \mathbf{a}R \in \mathbf{b}R$, but neither one of xb, xc, xd, or xe belongs to $\mathbf{b}(P \parallel R)$. So after x no progress is possible. Also $x \notin \mathbf{t}(P \parallel R)$ so we have:

$$edc \in \mathbf{deadlock}(P \parallel R)$$

The same can be said about the trace dec. □

Exercise 3.1 Which of the following connections are deadlock free:

(a) $\langle \{a,b\}, (ab) \rangle \parallel \langle \{a,b\}, (ba) \rangle$
(b) $\langle \{a,b\}, (ab)^* \rangle \parallel \langle \{a,b\}, (aba)^* \rangle$
(c) $\langle \{a,b\}, (aba) \rangle \parallel \langle \{a\}, a^* \rangle$

□

3.3 Livelock

Instead of ending in deadlock as described above it is also possible for a system to reach a point in behaviour from which no task can be completed, but which has the possibility that there is always a next event possible. This situation is called livelock. The system is locked in the sense that no task can be completed, but it is not deadlocked because always some event is possible.

Definition 3.3 *A system P is livelocked in $x \in \mathbf{b}(P)$ if*

$$(\forall y : y \in (\mathbf{a}P)^* \ \wedge \ xy \in \mathbf{b}P : xy \notin \mathbf{t}P \ \wedge \ (\exists a : a \in (\mathbf{a}P)^* : xya \in \mathbf{b}P))$$

The set of all livelocked traces is denoted by $\mathbf{livelock}(P)$
A system P is called livelock free, if

$$\mathbf{livelock}(P) = \emptyset$$

❐

In literature other interpretations of livelock can be found, for example the situation in which a system can repeat some part of its behaviour as many times as possible. Here we consider a system having the possibility of livelock, if it cannot reach a completed task any more, but is not deadlocked.

3.4 Lock

If a system is at some point in behaviour from which never a task can be completed, i.e., it may deadlock or it may livelock (see above), we speak of *livedeadlock*, or simply *lock*. It is defined as follows:

Definition 3.4 *A system P is* locked *in $x \in \mathbf{b}(P)$ if*

$$(\forall y : y \in (\mathbf{a}P)^* : xy \notin \mathbf{t}P)$$

The set of all locked traces is defined by

$$\mathbf{lock}(P) = \{x : x \in \mathbf{b}P \ \wedge \ (\forall y : y \in (\mathbf{a}P)^* : xy \notin \mathbf{t}P) : x\}$$

A system P is called lock free, notation $\mathbf{lockfree}(P)$*, if*

$$\mathbf{lock}(P) = \emptyset$$

❑

Because deadlock and livelock are special cases of lock, in the sequel we only consider locked systems.

Property 3.5

$$\mathbf{lockfree}(P) \Rightarrow \mathbf{livelockfree}(P) \ \wedge \ \mathbf{deadlockfree}(P)$$

❑

It is possible to get rid of the lock by adding a second system as is shown in the following example.

Example 3.6 Consider

$$P = \langle \{a, b, c, d\}, \mathbf{pref}(ad|abc), (abc) \rangle$$

then $\mathbf{lock}(P) = \{ad\}$. If we connect this system with $R = \langle \{b, d\}, b \rangle$ we find the connection

$$P \parallel R = \langle \{a, b, c, d\}, (abc) \rangle$$

that is free of lock. Notice that $R = \langle \{b\}, b \rangle$ leads to $P \parallel R = P$ so $\langle \{b\}, b \rangle$ does not remove the lock. ❑

3.4.1 Lock free subsystems

It is straightforward to find the greatest subsystem of some given DES P that is free of lock. It simply is the system

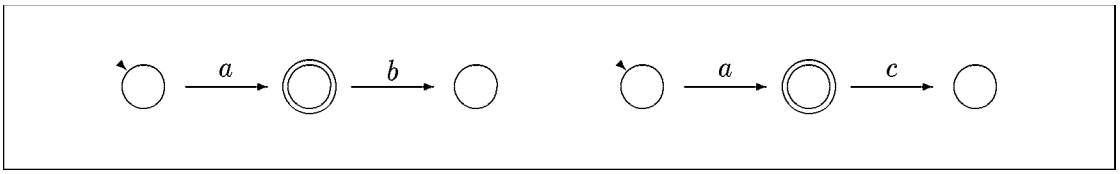
$$\langle \mathbf{a}P, \mathbf{pref}(\mathbf{t}P), \mathbf{t}P \rangle$$

Just delete all behaviour that does not lead to a completed task.

Exercise 3.2 Give lock-sets and the greatest lock free subsystems for:

(*a*) $\langle \{a, b, c\}, \mathbf{pref}(ab)^*, (ab) \rangle$
(*b*) $\langle \{a, b, c\}, \mathbf{pref}(ab|c)^*, (ab) \rangle$
(*c*) $\langle \{a, b, c\}, \mathbf{pref}(ab|c)^*, (abc) \rangle$

❑



Figuur 3.1: P (left) and R (right) for example 3.7

3.5 Lock free connections

We claim that, if some abitrary P and R have (in connection) the possibility of lock, we can construct a subsystem, say S, of R such that **lockfree**$(P \parallel S)$, where $S \subseteq R$. Of course, the system **empty**$(\mathbf{a}R)$ will always satisfy. However, we are searching for the largest possible S.

Notice that we do not impose any restrictions on the alphabets of P or R. In most cases, however, R controls P by using a subset of the events of P. The theory below assumes $\mathbf{a}P$ and $\mathbf{a}R$ to be arbitrary.

Example 3.7 First, notice that S itself need not be lock free in order to obtain a lock free connection with P. Consider P and R as given in figure 3.1 with $\mathbf{a}P = \mathbf{a}R = \{a, b, c\}$. Then

$$P \parallel R = \langle \{a, b, c\}, \mathbf{pref}(a), a \rangle$$

is lock free, but neither one of P and R is. □

In the sequel we try to find an algorithm that deletes some traces from the behaviour- and task set of a discrete system in order to get a lock free connection with a second system.

Simply deleting traces from the behaviour and task set may lead to a system that is no longer a DES. Therefore, we need the operator **real** on GDESs to get the DES-interior of it, which is the greatest subsystem that is a realistic DES, i.e., satisfies the properties of having a prefix closed behaviour and a task set that is part of the behaviour.

Definition 3.8 *If P is a discrete event system and T is some set of traces over alphabet $\mathbf{a}P$, then we define P* without *T as the system*

$$P \setminus\setminus T = \mathbf{real}(P \backslash T)$$

where $P \backslash T$ is an abbreviation for $P \backslash \langle \mathbf{a}P, T, T \rangle$. □

Example 3.9 If $P = \langle \{a, b, d\}, (\epsilon | a | b | ab | aba), (a | aba) \rangle$ and $T = \{ab\}$, then we find

$$\begin{aligned} P \backslash T &= \langle \{a, b, d\}, (\epsilon | a | b | aba), (a | aba) \rangle \\ P \setminus\setminus T &= \langle \{a, b, d\}, (\epsilon | a | b), a \rangle \end{aligned}$$

□

If P itself is realistic, we can define the operator $P \setminus\setminus T$ directly, without using the realistic interior, by

$$P \setminus\setminus T = \langle \mathbf{a}P, \mathbf{b}P \backslash (T(\mathbf{a}P)^*), \mathbf{t}R \backslash (T(\mathbf{a}P)^*) \rangle$$

It should be clear that this definition is equivalent to the previous one: instead of deleting the traces from T and creating holes in the trace sets (which are removed by computing the des-interior afterwards), you can also delete all traces from T together with all their extensions.

Example 3.9 (cont.)

$$T(\mathbf{a}P)^* \quad = \quad (ab(a|b|d)^*) = (ab|aba|abd|abaa|abab|\ldots)$$

Deleting this trace set from $\mathbf{b}P$ and $\mathbf{t}P$ directly gives the same result □

Example 3.10 Simply deleting all locked traces from one of the systems using this new operator will not do either. Consider

$$P = \langle \{a,b,c\}, (aca|aaba) \rangle \qquad R = \langle \{a\}, (a|aa) \rangle$$

then we have

$$P \parallel R = \langle \{a,b,c\}, \mathbf{pref}(aab|aca), (aca) \rangle$$

so $\mathbf{lock}(P \parallel R) = (aa|aab)$. Computing

$$S = R \setminus\setminus \mathbf{lock}(P \parallel R) \lceil \mathbf{a}R$$

results in $S = \langle \{a\}, a, \emptyset \rangle$ and

$$P \parallel S = \langle \{a,b,c\}, \mathbf{pref}(ac), \emptyset \rangle$$

which is, again, not free of lock. □

Our claim is that the following operator leads to the greatest possible lock free subsystem:

$$L(P,R) = R \setminus\setminus \mathbf{lock}(P \parallel R) \lceil \mathbf{a}R$$

Starting with $R_0 = R$ we can compute $R_{i+1} = L(P, R_i)$ and so find a chain of R_i's.

3.5.1 Some properties on the locked traces

We use the abbreviation $L_i = \mathbf{lock}(P \parallel R_i)$. The following properties can be derived, saying that the locked traces in the remaining connection are reduced in each next step in the chain.

Property 3.11

(1) $L_i \lceil \mathbf{a}R \subseteq \mathbf{b}R_i$
(2) $x \in L_{i+1} \lceil \mathbf{a}R \;\Rightarrow\; x \notin (L_i \lceil \mathbf{a}R)(\mathbf{a}R)^*$
(3) $x \in L_{i+1} \lceil \mathbf{a}R \Rightarrow (\exists v :: xv \in (L_i \lceil \mathbf{a}R)(\mathbf{a}R)^*)$

proof: (1) is trivial. For (2) we have:

$$
\begin{array}{ll}
 & x \in L_{i+1}\lceil \mathbf{a}R \\
\Rightarrow & \quad [\ (1)\ ] \\
 & x \in \mathbf{b}R_{i+1} \\
= & \quad [\ \text{construction of } R_{i+1}\ ] \\
 & x \in \mathbf{b}R_i \ \wedge\ x \notin (L_i\lceil \mathbf{a}R)(\mathbf{a}R)^*
\end{array}
$$

For (3) we have:

$$
\begin{array}{ll}
 & x \in L_{i+1}\lceil \mathbf{a}R \\
\Leftrightarrow & \quad [\ \text{definition of } \lceil\ ] \\
 & (\exists y : y \in L_{i+1} : y\lceil \mathbf{a}R = x) \\
\Leftrightarrow & \quad [\ \text{definition of } L_{i+1}\ ] \\
 & (\exists y : y \in \mathbf{b}(P \parallel R_{i+1}) : x = y\lceil \mathbf{a}R \ \wedge\ (\forall v :: yv \notin \mathbf{t}(P \parallel R_{i+1})) \\
\Leftrightarrow & \quad [\ y \in \mathbf{b}(P \parallel R_{i+1}) \Rightarrow y\lceil \mathbf{a}R \in \mathbf{b}R_{i+1} \Rightarrow y\lceil \mathbf{a}R \notin \mathbf{lock}(P \parallel R_i)\lceil \mathbf{a}R\ ] \\
 & (\exists y : y \in \mathbf{b}(P \parallel R_{i+1}) : x = y\lceil \mathbf{a}R \ \wedge\ x \notin (L_i\lceil \mathbf{a}R)(\mathbf{a}R)^* \ \wedge\ (\forall v :: yv \notin \mathbf{t}(P \parallel R_{i+1})) \\
\Rightarrow & \quad [\ R_{i+1} \subseteq R_i \text{ and } y\lceil \mathbf{a}R \notin L_i\lceil \mathbf{a}R \Rightarrow y \notin L_i\ ] \\
 & (\exists y : y \in \mathbf{b}(P \parallel R_i) : x = y\lceil \mathbf{a}R \ \wedge\ y \notin L_i \ \wedge\ (\forall v :: yv \notin \mathbf{t}(P \parallel R_{i+1})) \\
\Leftrightarrow & \quad [\ y \in \mathbf{b}(P \parallel R) \ \wedge\ y \notin L_i \Rightarrow (\exists v :: yv \in \mathbf{t}(P \parallel R_i)\ ] \\
 & (\exists y, v : x = y\lceil \mathbf{a}R : yv \in \mathbf{t}(P \parallel R_i) \ \wedge\ (\forall v :: yv \notin \mathbf{t}(P \parallel R_{i+1})) \\
\Rightarrow & \\
 & (\exists y, v : x = y\lceil \mathbf{a}R : yv \in \mathbf{t}(P \parallel R_i) \backslash \mathbf{t}(P \parallel R_{i+1})) \\
\Rightarrow & \quad [\ \text{definition of } \parallel \text{ and } \mathbf{t}R_i \backslash \mathbf{t}R_{i+1} \subseteq (L_i\lceil \mathbf{a}R)(\mathbf{a}R)^*\ ] \\
 & (\exists y, v : x = y\lceil \mathbf{a}R : yv\lceil \mathbf{a}R \in (L_i\lceil \mathbf{a}R)(\mathbf{a}R)^* \\
\Leftrightarrow & \\
 & (\exists v :: xv \in (L_i\lceil \mathbf{a}R)(\mathbf{a}R)^*
\end{array}
$$

□

Corollary 3.12

$$x \in L_{i+1}\lceil \mathbf{a}R \Rightarrow (\exists v : v \neq \epsilon : xv \in (L_i\lceil \mathbf{a}R))$$

□

This corrollary suggests that the chain of corresponding R_i's also decreases which means that the operator L has some greatest fix-point.

3.5.2 A fix-point solution of L

We claim that the operator L has a greatest fix-point and that this fix-point is the subsystem of R we are looking for.

The first property says that each iteration reduces the resulting subsystem and gives an important property of **lock**.

Property 3.13

(a) $L(P, R) \subseteq R$
(b) $\mathbf{lock}(P \parallel R)\lceil \mathbf{a}R \subseteq \mathbf{b}R$

□

If the fix-point exists it leads to a lock free connection with P:

Property 3.14

$$L(P,S) = S \Leftrightarrow \mathbf{lock}(P \parallel S) = \emptyset$$

proof:

$$\begin{array}{ll} & L(P,S) = S \\ \Leftrightarrow & [\text{ definition of operator } L \text{ }] \\ & S \setminus\setminus \mathbf{lock}(P \parallel S)\lceil \mathbf{a}S = S \\ \Leftrightarrow & [\ \mathbf{lock}(P \parallel S)\lceil \mathbf{a}S \subseteq \mathbf{b}S \ ] \\ & \mathbf{lock}(P \parallel S)\lceil \mathbf{a}S = \emptyset \end{array}$$

□

Lemma 3.15

$$\begin{array}{l} L(P,S) = S \ \wedge \ S \subseteq R \\ \Rightarrow \\ S \subseteq L(P,R) \end{array}$$

proof:

$$\begin{array}{ll} & \mathbf{b}S \subseteq \mathbf{b}(R \setminus\setminus \mathbf{lock}(P \parallel R)\lceil \mathbf{a}R) \\ = & [\text{ second definition of } \setminus\setminus \text{ }] \\ & \mathbf{b}S \subseteq \mathbf{b}R \backslash (\mathbf{lock}(P \parallel R)\lceil \mathbf{a}R)(\mathbf{a}R)^* \\ = & [\ S \subseteq R \ ] \\ & (\forall y, z : y \in \mathbf{lock}(P \parallel R) \ \wedge \ z \in (\mathbf{a}R)^* : (y\lceil \mathbf{a}R)z \notin \mathbf{b}S) \\ = & [\ \mathbf{lock}(P \parallel R) \subseteq \mathbf{b}(P \parallel R) \ ] \\ & (\forall y, z : y \in \mathbf{b}(P \parallel R) \ \wedge \ y \in \mathbf{lock}(P \parallel R) \ \wedge \ z \in (\mathbf{a}R)^* : (y\lceil \mathbf{a}R)z \notin \mathbf{b}S) \\ = & [\text{ trading }] \\ & (\forall y, z : y \in \mathbf{b}(P \parallel R) : z \notin (\mathbf{a}R)^* \ \vee \ (y\lceil \mathbf{a}R)z \notin \mathbf{b}S \ \vee \ y \notin \mathbf{lock}(P \parallel R)) \\ = & [\text{ nesting }] \\ & (\forall y : y \in \mathbf{b}(P \parallel R) : (\forall z :: z \notin (\mathbf{a}R)^* \ \vee \ (y\lceil \mathbf{a}R)z \notin \mathbf{b}S \ \vee \ y \notin \mathbf{lock}(P \parallel R))) \\ = & [\text{ distribution of } \vee \text{ over } \forall \text{ }] \\ & (\forall y : y \in \mathbf{b}(P \parallel R) : y \notin \mathbf{lock}(P \parallel R) \ \vee \ (\forall z :: z \notin (\mathbf{a}R)^* \ \vee \ (y\lceil \mathbf{a}R)z \notin \mathbf{b}S)) \\ = & [\text{ deMorgan }] \\ & (\forall y : y \in \mathbf{b}(P \parallel R) : y \notin \mathbf{lock}(P \parallel R) \ \vee \ \neg(\exists z :: z \in (\mathbf{a}R)^* \ \wedge \ (y\lceil \mathbf{a}R)z \in \mathbf{b}S)) \\ = & [\text{ trading }] \\ & (\forall y : y \in \mathbf{b}(P \parallel R) \ \wedge \ (\exists z :: z \in (\mathbf{a}R)^* \ \wedge \ (y\lceil \mathbf{a}R)z \in \mathbf{b}S) : y \notin \mathbf{lock}(P \parallel R)) \\ = & [\ \mathbf{b}S \text{ is prefix closed }] \\ & (\forall y : y \in \mathbf{b}(P \parallel R) \ \wedge \ y\lceil \mathbf{a}R \in \mathbf{b}S : y \notin \mathbf{lock}(P \parallel R)) \\ = & [\ S \subseteq R, \text{ and definitions of } \parallel \text{ and } \mathbf{lock} \ ] \\ & (\forall y : y \in \mathbf{b}(P \parallel S) : (\exists w :: yw \in \mathbf{t}(P \parallel R))) \\ \Leftarrow & [\ S \subseteq R \ ] \\ & (\forall y : y \in \mathbf{b}(P \parallel S) : (\exists w :: yw \in \mathbf{t}(P \parallel S))) \\ = & [\ (P \parallel S) \text{ is lock free }] \\ & \text{true} \end{array}$$

On the other hand:

$$\mathbf{t}S \subseteq \mathbf{t}(R \setminus\setminus \mathbf{lock}(P \parallel R)\lceil \mathbf{a}R)$$

$=$ [second definition of $\backslash\backslash$]
$\mathbf{t}S \subseteq \mathbf{t}R \backslash (\mathbf{lock}(P \parallel R) \lceil \mathbf{a}R)(\mathbf{a}R)^*$
$\Leftarrow$ [above result and R is a DES]
$\mathbf{t}S \subseteq \mathbf{t}R \ \wedge \ \mathbf{t}S \subseteq \mathbf{b}S$
$=$ [$S \subseteq R$ and S is a DES]
true

This combines to $S \subseteq R \backslash\backslash \mathbf{lock}(P \parallel R) \lceil \mathbf{a}R$. Notice that we have used here that the behaviour of both S and R is prefix closed and each task is a behaviour. □

We define the following set of classes of DESs:[1]

$$\Phi = \{\mathcal{V} : R \in \mathcal{V} \ \wedge \ (\forall S : S \in \mathcal{V} : L(P,S) \in \mathcal{V}) \ \wedge \ (\forall \mathcal{U} : \mathcal{U} \subseteq \mathcal{V} : \bigcap_{S \in \mathcal{U}} S \in \mathcal{V}) : \mathcal{V}\}$$

This set is not empty:

$$\{S : S \subseteq R : S\} \in \Phi$$

because:

(a) $R \in \{S : S \subseteq R : S\}$
(b) $(\forall S : S \subseteq R : L(P,S) \subseteq R)$ (see property 3.13 (a))
(c) $(\forall \mathcal{U} : \mathcal{U} \subseteq \{S : S \subseteq R : S\} : \bigcap_{S \in \mathcal{U}} S \subseteq R)$

Next, we define the intersection of all $\mathcal{V}$ from Φ to be the set $\mathcal{W}$:

$$\mathcal{W} = \bigcap_{\mathcal{V} \in \Phi} \mathcal{V}$$

We now have that this $\mathcal{W}$ is also a member of the class Φ:

Property 3.16

(a) $R \in \mathcal{W}$
(b) $(\forall S : S \in \mathcal{W} : L(P,S) \in \mathcal{W})$
(c) $(\forall \mathcal{U} : \mathcal{U} \subseteq \mathcal{W} : \bigcap_{S \in \mathcal{U}} S \in \mathcal{W})$

proof: Part (a):

true
$=$ [definition of Φ]
$(\forall \mathcal{V} : \mathcal{V} \in \Phi : R \in \mathcal{V})$
$=$
$R \in \bigcap_{\mathcal{V} \in \Phi} \mathcal{V}$

Part (b):

[1] These classes will not appear in the glossary, because they are only needed in this section to prove the result.

$$
\begin{array}{ll}
 & S \in \mathcal{W} \\
= & \quad [\text{ definition of } \mathcal{W} \text{ }] \\
 & S \in \bigcap_{\mathcal{V} \in \Phi} \mathcal{V} \\
= & \\
 & (\forall \mathcal{V} : \mathcal{V} \in \Phi : S \in \mathcal{V}) \\
\Rightarrow & \quad [\text{ definition of } \Phi, \text{ second property of } \mathcal{V} \text{ }] \\
 & (\forall \mathcal{V} : \mathcal{V} \in \Phi : L(P,S) \in \mathcal{V}) \\
= & \\
 & L(P,S) \in \bigcap_{\mathcal{V} \in \Phi} \mathcal{V} \\
= & \quad [\text{ definition of } \mathcal{W} \text{ }] \\
 & L(P,S) \in \mathcal{W}
\end{array}
$$

Part (c):

$$
\begin{array}{ll}
 & \mathcal{U} \subseteq \mathcal{W} \\
= & \quad [\text{ definition of } \mathcal{W} \text{ }] \\
 & \mathcal{U} \subseteq \bigcap_{\mathcal{V} \in \Phi} \mathcal{V} \\
= & \\
 & (\forall \mathcal{V} : \mathcal{V} \in \Phi : \mathcal{U} \subseteq \mathcal{V}) \\
\Rightarrow & \quad [\text{ definition of } \Phi, \text{ third property of } \mathcal{V} \text{ }] \\
 & (\forall \mathcal{V} : \mathcal{V} \in \Phi : \bigcap_{S \in \mathcal{U}} S \in \mathcal{V}) \\
= & \\
 & \bigcap_{S \in \mathcal{U}} S \in \bigcap_{\mathcal{V} \in \Phi} \mathcal{V} \\
= & \quad [\text{ definition of } \mathcal{W} \text{ }] \\
 & \bigcap_{S \in \mathcal{U}} S \in \mathcal{W}
\end{array}
$$

□

Now define

$$\Lambda(P,R) = \bigcap_{S \in \mathcal{W}} S$$

then we have that $\Lambda(P,R)$ is a fix-point of the operator L:

Property 3.17

$$\Lambda(P,R) = L(P, \Lambda(P,R))$$

proof: By construction of L we have $L(P,\Lambda(P,R)) \subseteq \Lambda(P,R)$, see property 3.13. Moreover:

$$
\begin{array}{ll}
 & \Lambda(P,R) = \bigcap_{S \in \mathcal{W}} S \\
\Rightarrow & \quad [\text{ see property 3.16 (c) with } \mathcal{U} = \mathcal{W} \text{ }] \\
 & \Lambda(P,R) \in \mathcal{W} \\
\Rightarrow & \quad [\text{ see property 3.16 (b) }] \\
 & L(P,\Lambda(P,R)) \in \mathcal{W}
\end{array}
$$

Hence

$$
\begin{array}{ll}
 & \Lambda(P,R) \\
= & \quad [\text{ definition }] \\
 & \bigcap_{S \in \mathcal{W}} S \\
\subseteq & \quad [\ L(P,\Lambda(P,R)) \in \mathcal{W}\ ] \\
 & L(P,\Lambda(P,R))
\end{array}
$$

□

The fix-point $\Lambda(P,R)$ leads to a lock free connection:

Lemma 3.18

$$\mathbf{lock}(P \parallel \Lambda(P,R)) = \emptyset$$

proof: from property 3.14 with $S = \Lambda(P,R)$ and property 3.17. □

Moreover, we have that $\Lambda(P,R)$ is the greatest fix-point, so it is the first one to be reached while computing the chain R_i:

Lemma 3.19

$$
\begin{array}{l}
S = L(P,S) \ \wedge \ S \subseteq R \\
\Rightarrow \\
S \subseteq \Lambda(P,R)
\end{array}
$$

proof: Let $\mathcal{V} = \{T : S \subseteq T \subseteq R : T\}$, then we have (a) $R \in \mathcal{V}$,

$$
\begin{array}{lll}
(b) & & T \in \mathcal{V} \\
 & \Rightarrow & \quad [\text{ definition of } \mathcal{V}\ ] \\
 & & S \subseteq T \subseteq R \\
 & \Rightarrow & \quad [\text{ lemma 3.15 with } L(P,S) = S, \text{ and } L(P,T) \subseteq T \subseteq R\ ] \\
 & & S \subseteq L(P,T) \subseteq R \\
 & \Rightarrow & \\
 & & L(P,T) \in \mathcal{V}
\end{array}
$$

$$
\begin{array}{lll}
(c) & & \mathcal{U} \subseteq \mathcal{V} \\
 & \Rightarrow & \\
 & & (\forall T : T \in \mathcal{U} : T \in \mathcal{V}) \\
 & \Rightarrow & \\
 & & \bigcap_{T \in \mathcal{U}} T \in \mathcal{V}
\end{array}
$$

(a), (b), and (c) give us that $\mathcal{V} \in \Phi$, and hence, by definition of $\mathcal{W}$, that $\mathcal{W} \subseteq \mathcal{V}$. Now we have

$$
\begin{array}{ll}
 & \Lambda(P,R) \\
= & \quad [\text{ definition }] \\
 & \bigcap_{S \in \mathcal{W}} S
\end{array}
$$

$\in$ [property 3.16 (c) with $\mathcal{U} = \mathcal{W}$]
$\mathcal{W}$
$\subseteq$ [above]
$\mathcal{V}$

So $\Lambda(P,R) \in \mathcal{V}$, that is $S \subseteq \Lambda(P,R) \subseteq R$, so $\Lambda(P,R)$ is the biggest one. □

We now have proved the following theorem:

Theorem 3.20 *$\Lambda(P,R)$ is the largest system contained in R for which the connection with P is free of lock.* □

From the above we have that the fix-point exists and has the required properties. Nothing is said about how many iterations are necessary to get $\Lambda(P,R)$. If P and R are regular it can be shown that this number is finite. Moreover, in that case we can prove that the fix-point can be effectively computed using finite state graphs or automatons. Both subjects will be discussed further on.

Example 3.21 If the fix-point is empty, no subsystem of R can be found such that the connection with P is free of lock. Consider P and R from example 3.10, then:

$$\begin{array}{lcll} R_0 & = & R & \\ R_1 & = & \langle\{a\},(a|aa)\backslash(aab|aa)\rangle & = \langle\{a\},a\rangle \\ R_2 & = & \langle\{a\},(a)\backslash(a|ac)\rangle & = \mathbf{empty}(\{a\}) \end{array}$$

and, indeed, no non-empty subsystem of R can be found with a lock free connection (just try every possible subsystem of R). □

3.5.3 Deadlock free connections

Actually, we can prove the same for the deadlock case. Use

$$\Delta(P,R) = \bigcap_i R_i \quad \text{with} \begin{cases} R_0 = R \\ R_i = R_{i-1} \setminus\setminus \mathbf{deadlock}(P \parallel R_{i-1}) \lceil \mathbf{a}R & i = 1,2,\ldots \end{cases}$$

Then we have:

Theorem 3.22 *$\Delta(P,R)$ is the largest system contained in R for which the connection with P is free of deadlock.* □

Exercise 3.3 Given $P = \langle\{a,b,c\}, \mathbf{pref}(ab|c), ab\rangle$ find the greatest controller R such that $P \parallel R$ is free of lock. Try each of the following alphabets for R:

(a) $\mathbf{a}R = \{a,b\}$
(b) $\mathbf{a}R = \{a,c\}$
(c) $\mathbf{a}R = \{a,b,c\}$

□

3.6 References

The theory presented here first appeared in [Sme91] and [Sme90a]. Another definition of deadlock is given by Kaldeway in [Kal88]. He defines a combination of systems to be in deadlock if any subcombination of these systems causes deadlock. This definition makes it impossible to get rid of the deadlock by adding an extra system. In the definition presented in this chapter, deadlock (or lock) is only considered on the whole combination of systems. Therefore, it is possible to get rid of the deadlock by adding one extra system to the combination.

Also in the framework of Ramadge and Wonham there exists theory in finding so called nonblocking supervisors, i.e. a supervisor that not only leads to the desired behaviour but also to a lock free connection (see [LW88, RW89]).

The nonblocking property is an extra demand on the supervisor. In our approach lock free (or nonblocking) is a demand on its own. Given some plant P we can construct a controller R that leads in connection with P to a lock free system. Our approach is more general than finding a nonblocking supervisor because (a) no restrictions are put on the system P that should be controlled (we only need that P has a realistic interpretation, i.e., $\mathbf{pref}(\mathbf{b}P) = \mathbf{b}P$ and $\mathbf{t}P \subseteq \mathbf{b}P$), especially no modelling of the behaviour in some way is necessary (no automatons), and (b) getting a lock free connection is a demand on its own (and not part of another control problem like with the nonblocking supervisor).

Thanks go to Wim Hesselink for reading earlier drafts of this part and for finding a more elegant proof for lemma 3.15 and for suggestions that lead to the proof of theorem 3.20.

4

Locked systems in state space

In this chapter we give algorithms on state graphs for determining lock and for finding the greatest lock free subprocess of a given process. It turns out that all algorithms given here can be computed effectively. i.e., can be done in a finite number of steps if the graphs to start with are regular.[1]

4.1 State graph for trace exclusion

From definition 3.8 we have

$$\mathbf{des}(\mathbf{real}(\mathbf{sg}(P)\backslash\mathbf{sg}(\langle \mathbf{a}P, T, T\rangle))) = P \setminus\setminus T$$

Computing $\mathbf{sg}(P \setminus\setminus T)$ can be reduced (as can all of the computations on state graphs) if we only consider reachable states: i.e., start from the new initial state and only consider those states, reachable from already computed states.
Computing $\mathbf{real}(\mathbf{sg}(P)\backslash\mathbf{sg}(R))$ can be reduced even more: compute $\mathbf{sg}(P)\backslash\mathbf{sg}(R)$ in the normal way, but, as soon as a non-task/non-behaviour state is reached no further computations in that direction are needed. In fact all arrows going to such a state can be redirected to one (fresh) dump state, thus creating the realistic interior in one go.

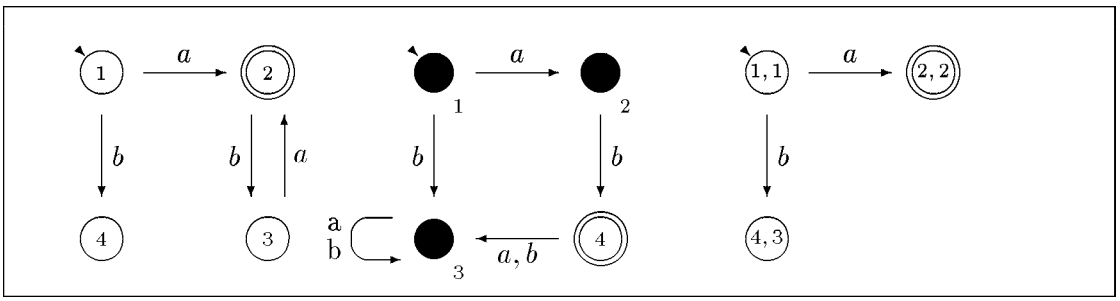
Example 4.1 Consider P and T as given in figure 4.1. The graph $\mathbf{real}(\mathbf{sg}(P)\backslash\mathbf{sg}(T))$ is also given in figure 4.1. Indeed, this graph represents the system $P \setminus\setminus T$. □

Exercise 4.1 Consider

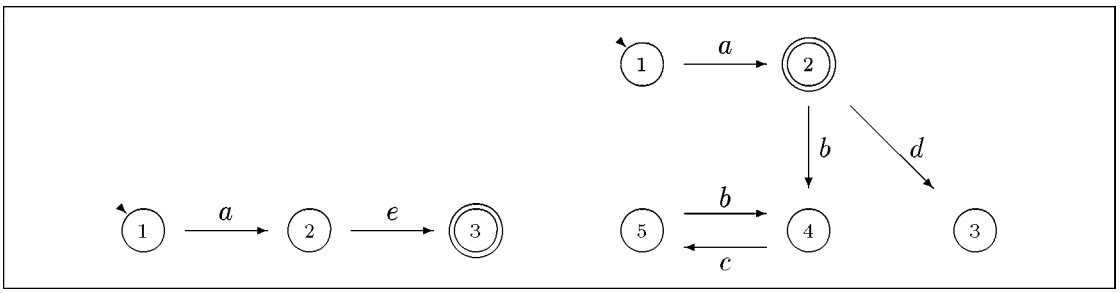
$$\begin{aligned} P &= \langle \{a, b\}, (a|b)^* \rangle \\ T &= ab|ba \end{aligned}$$

Construct $\mathbf{sg}(P)$ and compute a state graph for $P \setminus\setminus T$. Check that the state graph for $P \setminus\setminus T$ can be much larger than that for P. □

[1] Although the theory on state graphs as is presented here, holds for all graphs, especially those representing a possibly non-realistic DES, the correspondence to the previous chapter only holds for realistic DESs.



Figuur 4.1: $\mathbf{sg}(P)$ *(left),* $\mathbf{sg}(T)$ *(middle), and* $\mathbf{real}(\mathbf{sg}(P) \backslash \mathbf{sg}(T))$ *(right) for example 4.1*



Figuur 4.2: State graphs for P (left) and R (right) of example 4.6

4.2 Detecting Lock

4.2.1 Deadlock

Deadlock can simply be detected in the corresponding state graph of P by searching for deadlock states, i.e. states with no outgoing arrows, except for those pointing to the non behaviour state:

Definition 4.2 *A state* $p \in \mathbf{B}(P)$ *is called a* deadlock state *if*

$$(p \notin \mathbf{T}(P) \ \wedge \ (\forall a : a \in \mathbf{a}P : \delta(p, a) \notin \mathbf{B}(P))$$

❑

Lemma 4.3 *For a DES P we have:*

$$x \in \mathbf{deadlock}(P)$$

=

$$[x] \textit{ is a deadlock state in } \mathbf{sg}(P)$$

❑

4.2.2 Lock

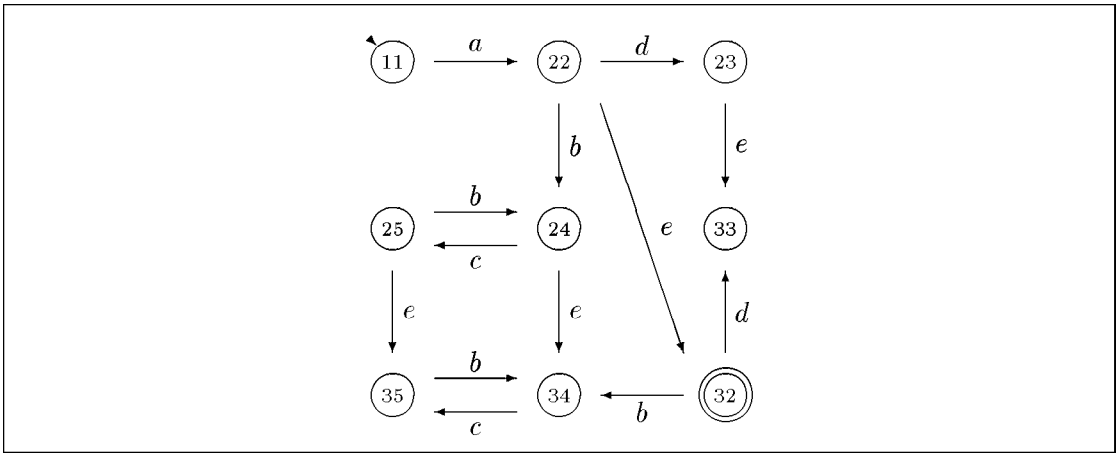
In this section we give (in case P is regular) a method to detect lock. Therefore, we need the corresponding finite state graph $\mathbf{sg}(P)$. In case we deal with connections we first have to find the product automaton $\mathbf{sg}(P \parallel R)$ for the connection $P \parallel R$.

To be able to use $\mathbf{sg}(P)$ for detecting lock we also need the next definition:

Definition 4.4 *A state* $p \in \mathbf{B}(P)$ *is called a* locked state *if:*

$$(\forall y : y \in (\mathbf{a}P)^* : \delta(p, y) \notin \mathbf{T}(P))$$

❑



Figuur 4.3: Corresponding product graph $\mathbf{sg}(P \parallel R)$ *from example 4.6*

Lemma 4.5 *For a DES* P *we have:*

$x \in \mathbf{lock}(P)$

=

$[x]$ *is a locked state in* $\mathbf{sg}(P)$

□

Example 4.6 Consider the processes P and R with $\mathbf{sg}(P)$ and $\mathbf{sg}(R)$ given in figure 4.2 and $\mathbf{a}P = \{a, e\}$ and $\mathbf{a}R = \{a, b, c, d\}$. The constructed $\mathbf{sg}(P \parallel R)$ is as in figure 4.3. We see that all states but 11, 22, and 32 are locked states in this product graph. State 33 is also a deadlock state (no starting arrows). Also, we have $\neg\mathbf{lockfree}(R)$ because states 3, 4, and 5 are locked states in $\mathbf{sg}(R)$. □

4.2.3 Detecting locked states

We use reversed graphs to detect the locked states in the following way: first we reverse all arrows of the state graph, next we determine which behaviour states are reachable from a task state in this reversed graph (these states have paths going to a task state in the original state graph). All states that cannot be reached are therefore locked states.

Because the labels along the arrows are not important, nor are the initial states in this graph, we use a degenerated graph with no alphabet and no initial state. Therefore the transition reduces to a function $\overline{\delta} : Q \to \mathbf{2}^Q$, or, equivalently, $\overline{\delta} \subseteq Q \times Q$. Formally:

Definition 4.7 *For* $\mathbf{sg}(P)$*, the* reversed graph*, denoted* $\mathbf{rev}(P)$*, is defined by*

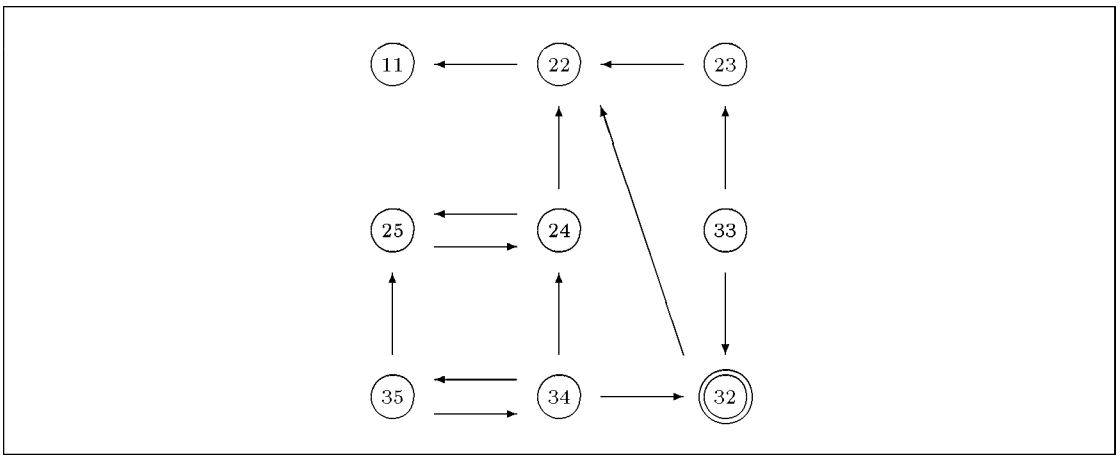
$\mathbf{rev}(P) = (\mathbf{B}(P), d, \mathbf{T}(P))$

where $d\colon \mathbf{B}(P) \to \mathbf{2}^{\mathbf{B}(P)}$ *is defined by*

$q \in d(p) \Leftrightarrow (\exists a : a \in \mathbf{a}P : \delta(q, a) = p)$

□

Example 4.8 Using this construction on $\mathbf{sg}(P \parallel R)$ from example 4.6 we find the reversed graph $\mathbf{rev}(P \parallel R)$ as given in figure 4.4. □



Figuur 4.4: Reversed graph of figure 4.3

Definition 4.9 *A state q is reachable from a state p in* $\mathbf{rev}(P)$ *if*

$$q = p \ \vee \ q \text{ is reachable from a state } \overline{q} \in d(p)$$

The set of all reachable states from state p is denoted by $\mathbf{reach}(p)$. □

It is easy to see that

$$\mathbf{reach}(p) = \{q : (\exists x : x \in (\mathbf{a}P)^* : \delta^*(q, x) = p)) : q\}$$

so $\mathbf{reach}(p)$ contains all states in $\mathbf{sg}(P)$ from which a path to p exists. $\mathbf{reach}(p)$ contains all states from which p can be reached. According to definition 4.4 we see that the set of all locked states is equal to $\mathbf{B}(P)\backslash(\mathbf{reach}(\mathbf{F}(P)))$.[2] This leads to the following algorithm to find the locked states of process P:

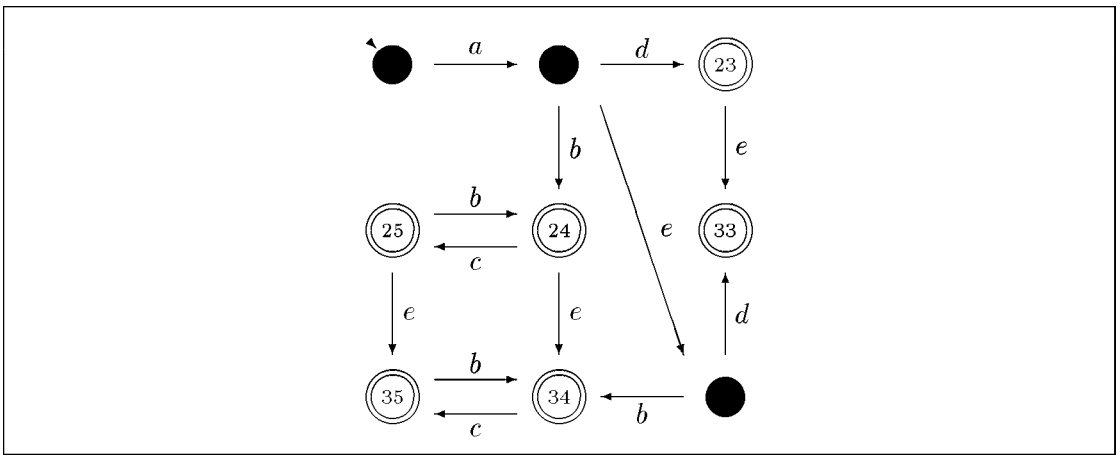
1) construct $\mathbf{rev}(P)$
2) collect the reachable states $\mathbf{reach}(\mathbf{F}(P))$
3) the set $\mathbf{B}(P)\backslash(\mathbf{reach}(\mathbf{F}(P)))$ is the set of all locked states in $\mathbf{sg}(P)$

The algorithm can be programmed easily.

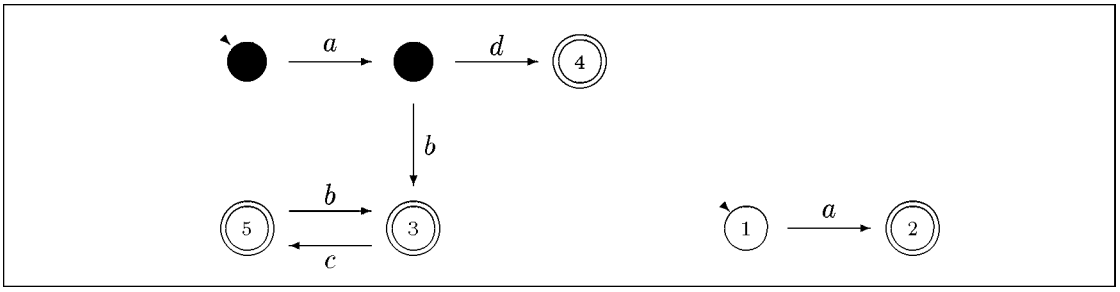
Example 4.10 In the previous example the states 32, 22, and 11 are in $\mathbf{reach}(q_{32})$. The other states (23, 24, 25, 33, 34, 35) are not and therefore are the locked states of the process P. □

Exercise 4.2 Compute a state graph for $P = \langle\{a, b, c\}, \mathbf{pref}(ab|c)^*, (abc)\rangle$ and compute, using the above method $\mathbf{sg}(\mathbf{lock}(P))$. □

[2] Where $\mathbf{reach}(\mathbf{F}(P)) = \bigcup_{p \in \mathbf{F}(P)} \mathbf{reach}(p)$.



Figuur 4.5: Graph $\mathbf{sg}(\mathbf{lock}(P \parallel R))$ *for example 4.11*



Figuur 4.6: Left: Graph for $\mathbf{lock}(P \| R)\lceil \mathbf{a}R$*; Right: Graph for* $R \setminus\setminus \mathbf{lock}(P \| R)\lceil \mathbf{a}R$*; see example 4.11*

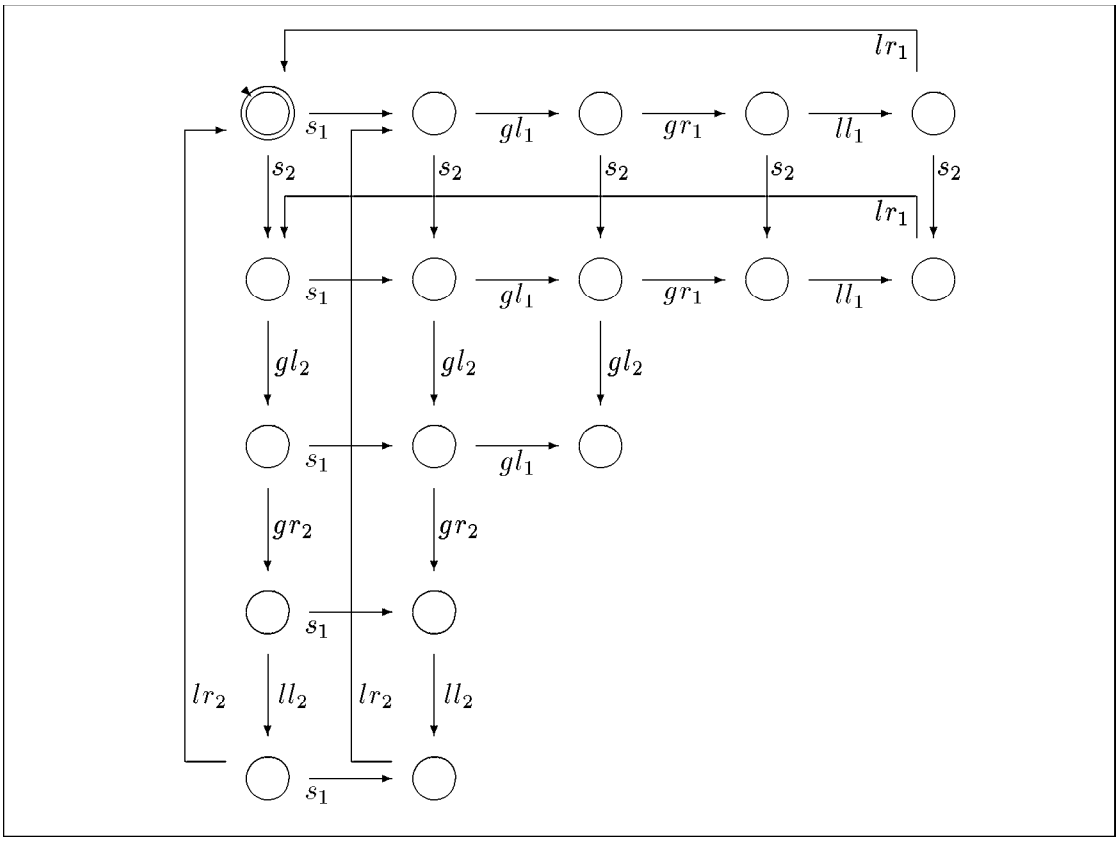
4.3 Lock free connections

If we use regular discrete processes, i.e., represented by finite state graphs, the algorithm to find $\Lambda(P, R)$ (and similar to find $\Delta(P, R)$) can be programmed on the finite state graphs, as is shown below.

The state graph of $\mathbf{lock}(P \parallel R)$ can easily be found using the product graph of P and R in which all states are changed into non-behaviour states and all locked states are changed into behaviour/task states. Here, we use the method of section 4.2.3 to determine the locked states. Changing all arrows with labels not in $\mathbf{a}R$ into arrows labeled with ϵ then yields an nd-graph that represents $\mathbf{lock}(P \parallel R)\lceil \mathbf{a}R$. After making this graph deterministic using **det**, we can again use the minus graph to find the graph for $R \setminus\setminus \mathbf{lock}(P \parallel R)\lceil \mathbf{a}R$. The following example illustrates this.

Example 4.11 Reconsider P and R from example 4.6. Then $\mathbf{lock}(P \| R)$ is represented by the graph in figure 4.5. Figure 4.6 (left) gives the graph for $\mathbf{lock}(P \| R)\lceil \mathbf{a}R$, after using property 2.14 to make it deterministic, and figure 4.6 (right) shows the corresponding minus graph. □

Summarized:
Given are $\mathbf{sg}(P)$ and $\mathbf{sg}(R)$, then



Figuur 4.7: The two dining philosophers

1) compute $\mathbf{sg}(P \parallel R)$
2) compute a graph for $\mathbf{lock}(P \parallel R)$ using $\mathbf{rev}(PwR)$
3) compute the graph G for $\mathbf{lock}(P \parallel R)\lceil \mathbf{a}R$ and make it deterministic again
4) compute the graph $\mathbf{real}(\mathbf{sg}(P)\backslash G)$
5) repeat this sequence until two consequtive results are the same.

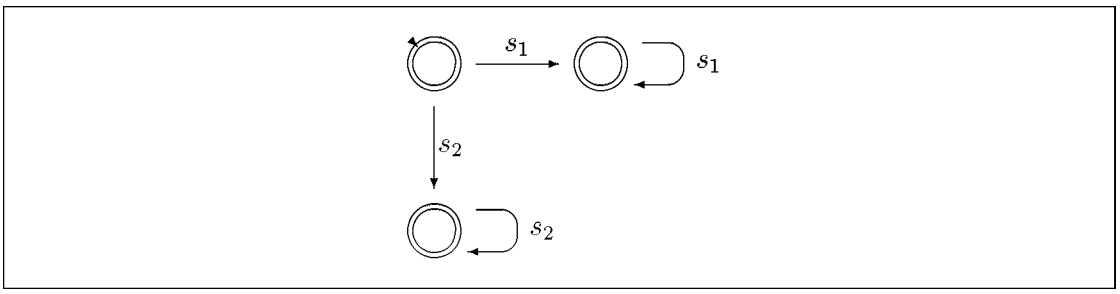
4.4 An example

In this chapter we discuss the well known problem of the dining philosophers in order to illustrate the use of the greatest subsystem. In order to keep things simple we only consider the situation of two philosophers, each with behaviour given by:

$$\mathbf{t}phil_i = (s_i \cdot gl_i \cdot gr_i \cdot e_i \cdot ll_i \cdot lr_i \cdot t_i)^* \qquad i = 1, 2$$

with:

s_i	permission to sit down
gl_i	grap left fork
gr_i	grap right fork
ll_i	lay down left fork
lr_i	lay down right fork
t_i	think



Figuur 4.8: The first butler

The events e_i and t_i will not further be modelled.

Each of the two philosophers needs two forks to eat. A fork can only be in use by one of the philosophers, so a fork should first be layed down, if it is in use by the other philosopher, before it can be used again. We use the following addition systems to model this:

$$\mathbf{t}fork_i = ((gl_i \cdot ll_i)|(gr_{i\oplus 1} \cdot lr_{i\oplus 1}))^*$$

were $\oplus$ denotes addition modulo 2.

Our total system under consideration now becomes:

$$P = phil_1 \parallel phil_2 \parallel fork_1 \parallel fork_2$$

The behaviour of this system is displayed in figure 4.7.

We see that there is a possibility of lock (deadlock in this case) if both philosophers have taken one fork and are waiting for the other fork to be released. In literature a solution is found using a butler. We use the results from the previous chapter to find such a butler, i.e., we are looking for some system *butler* such that **deadlockfree**$(P \parallel butler)$.

First, we notice that s_1 and s_2 can be used by the butler to control the whole (these are the only events that can be influenced by the environment, all other events are performed solely by the philosophers). Moreover, we start with a butler as general as possible, i.e., one who permits everything, so we start with

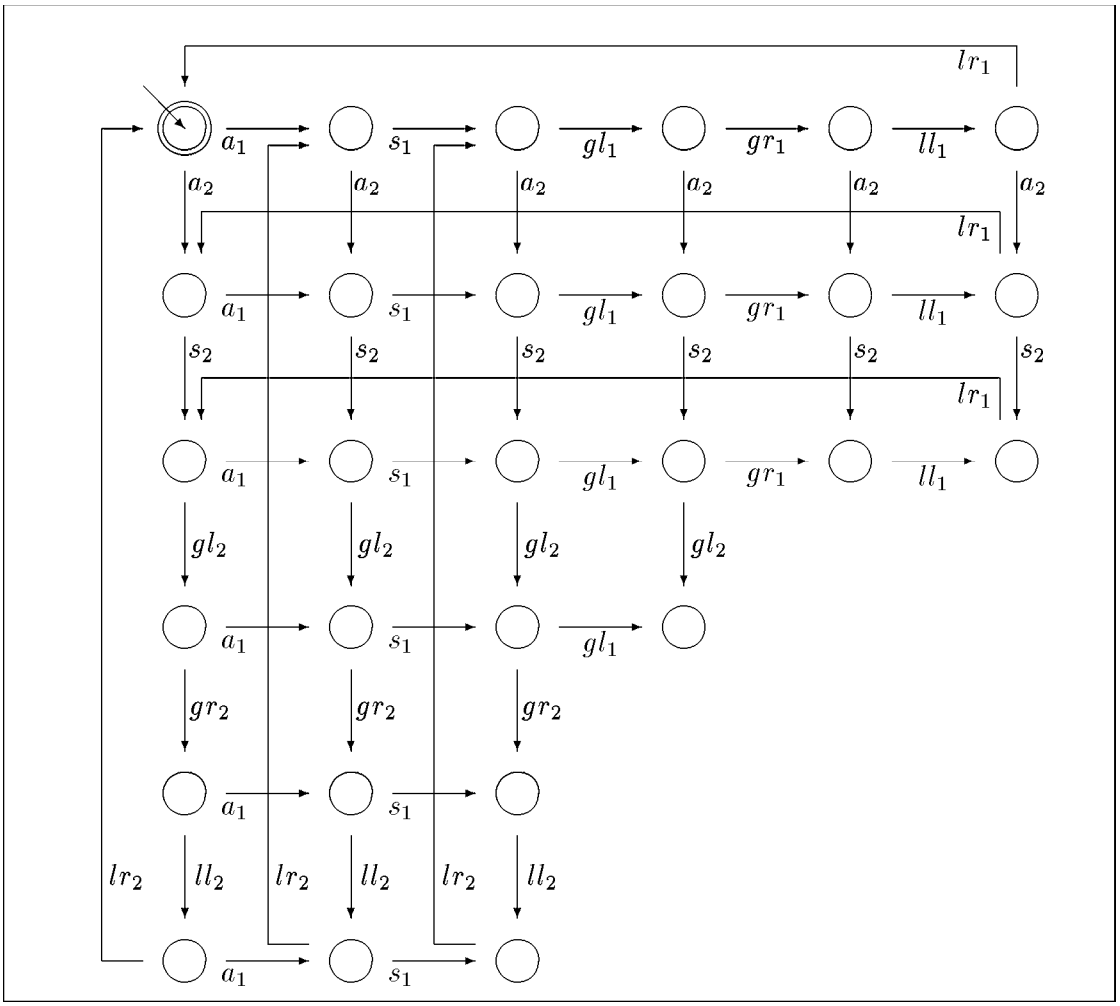
$$R_0 = \langle \{s_1, s_2\}, (s_1|s_2)^* \rangle$$

Starting with such an R_0 means that we demand minimal restrictions on the system as a whole. The resulting butler is the greatest subsystem of this R_0 and, because R_0 is chosen as large as possible, the butler will cause as less restrictions as possible on the total system. Starting with such a general R_0 means finding the most general system that omits deadlock.

We compute

$$R_{i+1} = R_i \setminus\setminus \mathbf{deadlock}(P \parallel R_i)|\mathbf{a}R$$

until a fix-point is reached. We find $butler = \Delta(P, R_0) = R_1$, given in figure 4.8. The butler only gives one (randomly chosen) philosopher permission to eat. This solution is not very elegant but it is everything we can do to omit deadlock.



Figuur 4.9: The two extended philosophers

We add one extra event to the system and use

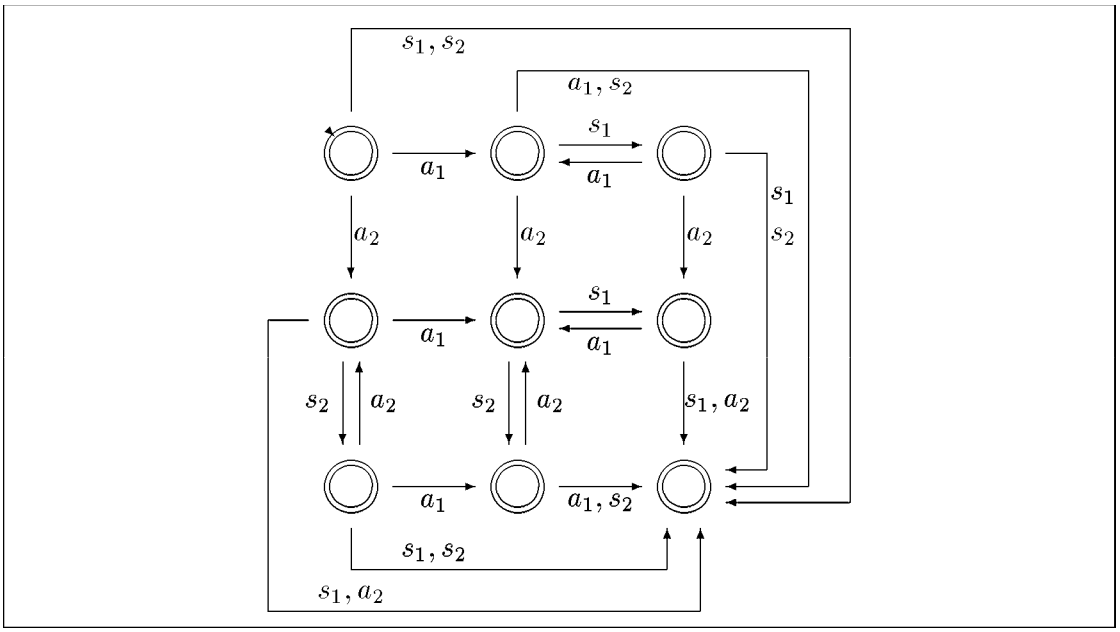
$$\mathbf{t}phil_i = (a_i \cdot s_i \cdot gl_i \cdot gr_i \cdot ll_i \cdot lr_i)^* \qquad i = 1, 2$$

with: a_i meaning "ask permission to sit down." The total system becomes as displayed in figure 4.9.

Now we start with

$$R_0 = \langle \{a_1, a_2, s_1, s_2\}, (s_1|s_2|a_1|a_2)^* \rangle$$

and find $butler = \Delta(P, R_0) = R_1$ as displayed in figure 4.10. Part of the behaviour of the butler is superfluous and can be omitted without changing the interaction with the total system. We can use $butler \cap P\lceil \mathbf{c}P$ instead, which we call the *effective part* of the controller (see figure 4.11). We see that this extra system permits one philosopher to sit and eat. A second demand is retrained until the first philosopher is ready and asks again to sit down. If we display the resulting behaviour of butler in connection with the philosophers and fork-processes we find the system as is displayed in figure 4.12.



Figuur 4.10: Second butler

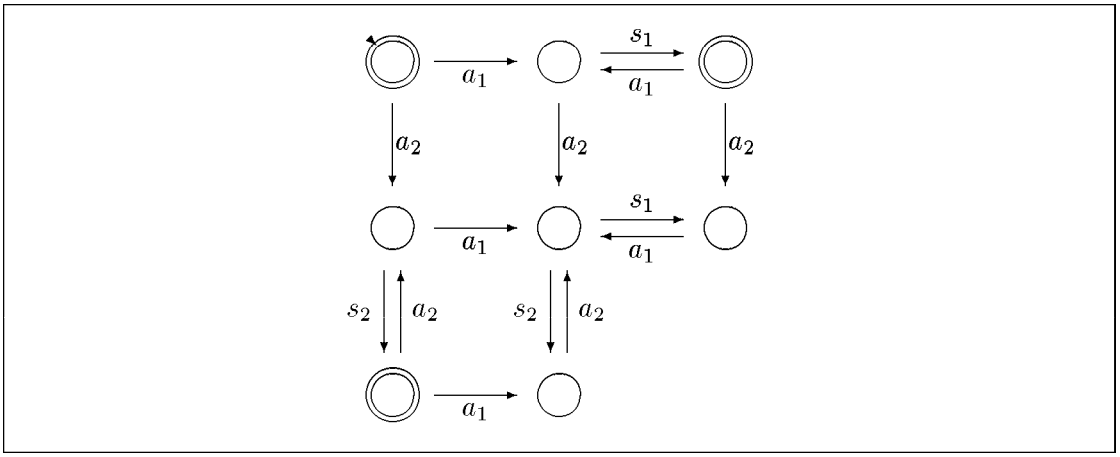
The butler does not omit livelock! For example, after behaviour $a_1 a_2$ no task state can be reached any more. If we use the algorithm to find a butler that omits livelock as well, so compute $\Lambda(P, R_0)$ instead, we find the butler as is displayed in figure 4.13 and see that, again, we have a butler that only permits one philosopher to eat and think. Because we start with an R_0 as general as possible we can conclude that no butler can be found also that (within the given configuration) omits livelock and does not cause starvation.

Exercise 4.3 Recall exercise 4.2. Compute the greatest controller R with $\mathbf{a}R = \{a, c\}$ such that $P \parallel R$ is free of lock. □

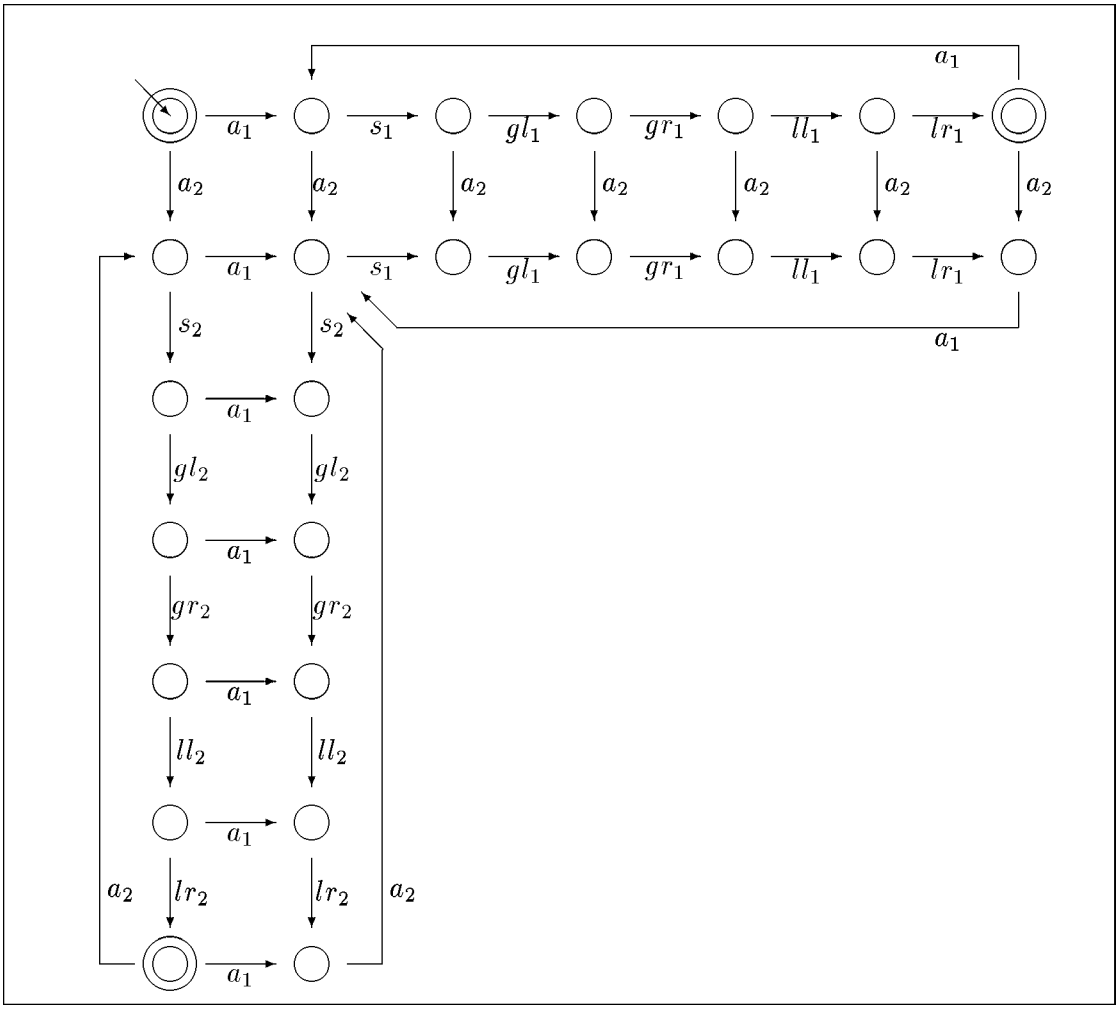
Exercise 4.4 Consider P and R as given in figure 4.14. Compute the fix-point of $L(P, R)$. □

References

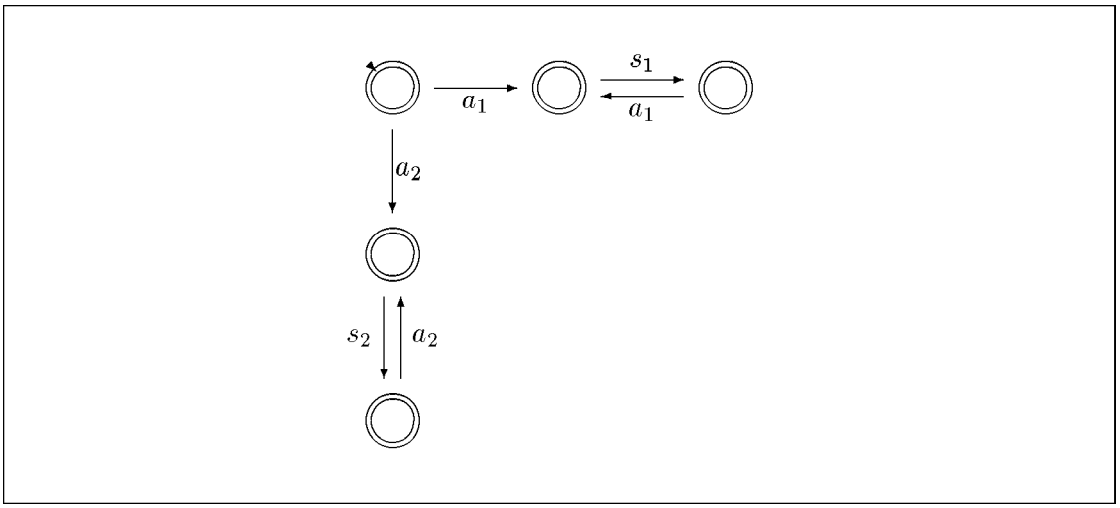
The material presented in this chapter can also be found in two technical reports: [Sme90a] and [Sme90b].



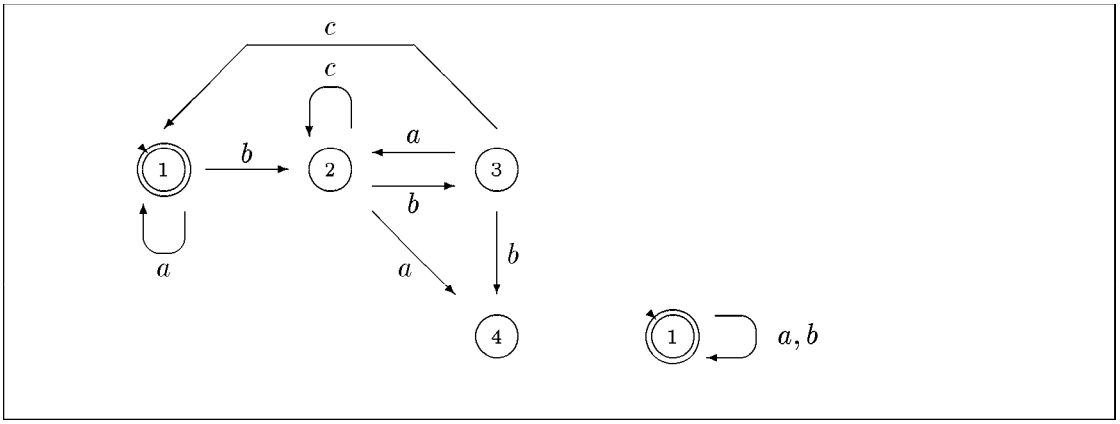
Figuur 4.11: Effective part of second butler



Figuur 4.12: Resulting behaviour with second butler



Figuur 4.13: A third butler



Figuur 4.14: Left: P; Right: R; for exercise 4.4

5
Control problems

In this chapter we introduce control problems on discrete event systems. Control of a discrete event system means using some of the events to control the order of the remaining events.[1]

In order to be able to introduce control we split the alphabet while considering two kinds of events:

- exogenous events
- communication events

The exogenous events are used to model actions introduced by the systems own dynamics. This means that exogenous events do not appear in other systems.[2] The communication events are used to model actions of a system, that may be common to other systems. This kind of event is of interest in communication with other systems and will be used to control the exogenous events.

Example 5.1 Consider the following lock free doctor-system:

$$P = \langle \{a, d, e\}, (aed)^* \rangle$$

with a meaning *arrival of a patient in the doctor's office*, e meaning *treat a patient*, and d meaning *departure of a patient*. The arrival and departure of patients are communication events (a patient can only arrive if the environment (i.e., some other system) "supplies" one; also, a patient can only depart if the environment can accept one). The treatment of the patient, however, is done by the doctor system P itself. It cannot be influenced by the environment. So $\{a, d\}$ is the set of communication events of P and $\{e\}$ is the set of exogenous events of P. Of course, in general, the answer to the question if an event is an exogenous event or an communication event depends not only on the interpretation of the system but also on the control problem to be solved. □

[1] In fact, other interpretations of control are possible. Here, we consider ordering as the most important aspect of discrete event systems and therefore control of systems should mean control of the order of (a number of) events of the system.

[2] One can argue about the name *exogenous*. *Endogenous* events would perhaps be more convenient (endogenous (exogenous) means growing or originating from within (without)). However it turns out that exogenous events are retained in a external connection while communication events disappear. So communication events are internal and exogenous events are external with respect to connection. Moreover, exogenous is a standard term in system theory, although it is used in a slightly different setting here.

According to our control problem we suppose it is possible to divide the alphabet of some system P into two subsets, denoted by $\mathbf{c}P$ and $\mathbf{e}P$, such that

$$\mathbf{c}P \cup \mathbf{e}P = \mathbf{a}P \qquad \mathbf{c}P \cap \mathbf{e}P = \emptyset$$

$\mathbf{c}P$ will be called the communication alphabet of P and $\mathbf{e}P$ will be called the exogenous alphabet of P.

5.1 The control problem

In the sequel we will deal with the following control problem:

Given is a discrete event system $P = \langle \mathbf{a}P, \mathbf{b}P, \mathbf{t}P \rangle$ with $\mathbf{e}P \subseteq \mathbf{a}P$ and $\mathbf{c}P = \mathbf{a}P \setminus \mathbf{e}P$. Moreover discrete event systems $L_{\min}$ and $L_{\max}$ are given with[3]

$$L_{\min} \subseteq L_{\max} \subseteq P\lceil \mathbf{e}P$$

$L_{\min}$ and $L_{\max}$ specify the range of resulting exogenous behaviour and exogenous tasks that are acceptable.
The problem is to find, if possible, a discrete system R with:

$$R = \langle \mathbf{a}R, \mathbf{b}R, \mathbf{t}R \rangle \qquad \text{with } \mathbf{a}R \subseteq \mathbf{c}P$$

such that

$$L_{\min} \subseteq (P \parallel R)\lceil \mathbf{e}P \subseteq L_{\max}$$

This last condition is called the *minmax condition.*
In the sequel this problem is referred to as CODE.

Notice that, if $\mathbf{a}R = \mathbf{c}P$, the minmax condition can also be written as

$$L_{\min} \subseteq P \rceil\lceil R \subseteq L_{\max}$$

Example 5.2 A possible control problem could be:
Given

$$\begin{array}{lcl} P & = & \langle \{a,b,c,d,e,g\}, \mathbf{pref}(ae|ad|ag|be|cg), (ae|be|cg) \rangle \\ \mathbf{e}P & = & \{d,e,g\} \\ \mathbf{c}P & = & \{a,b,c\} \\ L_{\min} & = & \langle \{d,e,g\}, (e) \rangle \\ L_{\max} & = & \langle \{d,e,g\}, (e|g) \rangle \end{array}$$

find a controller R with $\mathbf{a}R = \{a,b\}$ such that $L_{\min} \subseteq P \rceil\lceil R \subseteq L_{\max}$. □

Example 5.3 Suppose a shop sells two kinds of articles and in order to get an article one has to pay for it. Paying for an article is supposed to be a communication action. So we have as events:

a_1 sell article 1
a_2 sell article 2
p_1 pay for article 1
p_2 pay for article 2

[3] Notice, that this assumption implies that $\mathbf{a}L_{\min} = \mathbf{a}L_{\max} = \mathbf{e}P$.

The complete shop system becomes

$$P = \langle \{b_1, b_2, p_1, p_2\}, ((p_1 b_1)|(p_2 b_2))^* \rangle$$

The behaviour of P is a repetitive choice of paying for article 1 (p_1) and buying it (b_1) and paying for article 2 (p_2) and buying it (b_2). For example $p_1 b_1 p_1 b_1 p_2 b_2 p_1 b_1$ is a legal task of P. A customer can now be described as "pay for every article wanted," for example:

$$R = \langle \{p_1, p_2\}, ((p_1 p_1), p_2) \rangle$$

if the customer wants to have two articles number 1 and one article number 2. In fact, R is a controller for the shop system: it controls the events b_1 and b_2 using the events p_1 and p_2. If we consider $\mathbf{e}P = \{b_1, b_2\}$ then we have

$$(P \parallel R)\lceil \mathbf{e}P = \langle \{b_1, b_2\}, ((b_1 b_1), b_2) \rangle$$

Notice that the uncontrolled task set of the shop P equals $\mathbf{t}P\lceil \mathbf{e}P = (b_1|b_2)^*$, while the controlled task set (after connection with R) is $\mathbf{t}(P \parallel R)\lceil \mathbf{e}P = (b_1 b_1), b_2$.

A corresponding control problem could be to buy at most two articles of kind 2 and always one more of kind 1. The desired $L_{\min}$ and $L_{\max}$ then are

$$L_{\min} = \langle \{b_1, b_2\}, (b_1) \rangle$$
$$L_{\max} = \langle \{b_1, b_2\}, (b_1|(b_1 b_1, b_2)|(b_1 b_1 b_1, b_2 b_2)) \rangle$$

The R above turns out to be a solution for this problem, because:

$$L_{\min} \subseteq (P \parallel R)\lceil \mathbf{e}P \subseteq L_{\max}$$

□

In conventional system theory a system is controlled using the outputs as observations and computing new inputs for the system in order to get a desired behaviour (for example to make the system stable or decouple disturbances). In fact, the basic idea here is the same: we use the communication events[4] to establish some predefined behaviour. The main difference is of course that our desired behaviour has to do with order of events, not with behaviour in time.

5.2 Solution for CODE

5.2.1 Assumption and notation

In the sequel we deal (without loss of generality)[5] only with the situation that $\mathbf{a}R = \mathbf{c}P$. So we use *all* communication events to control the system. The control problem can then be easily written as

$$L_{\min} \subseteq P \parallel R \subseteq L_{\max}$$

The control problem can also be viewed as a design problem: P is a first design guess, R is needed to complete the design.

[4] At this point no distinction is made between inputs and outputs.
[5] In section 5.7 we discuss more general settings of CODE.

5.3 Solution for the control problem

We claim that the following DES leads to a solution of the control problem.

$$F(P,L) = {\sim}(P \rceil\lceil {\sim}L)$$

$F(P,L)$ will be called the *friend* of P and L. The resulting connection will be denoted by $G(P,L)$, i.e.,

$$G(P,L) = P \,\|\, F(P,L)$$

$G(P,L)$ sometimes will be called the *guardian*. First, we need two properties concerning the reflection.

Property 5.4

$$P \subseteq R \Leftrightarrow P \,\|\, {\sim}R = \mathbf{empty}(\emptyset)$$

proof:

$$\begin{array}{ll} & P \subseteq R \\ \Leftrightarrow & \text{[definition of } \subseteq \text{]} \\ & \mathbf{a}P = \mathbf{a}R \ \wedge\ \mathbf{b}P \subseteq \mathbf{b}R \ \wedge\ \mathbf{t}P \subseteq \mathbf{t}R \\ \Leftrightarrow & \text{[set theory]} \\ & \mathbf{a}P = \mathbf{a}R \ \wedge\ \mathbf{b}P \cap ((\mathbf{a}R)^* \backslash \mathbf{b}R) = \emptyset \ \wedge\ \mathbf{t}P \cap ((\mathbf{a}R)^* \backslash \mathbf{t}R) = \emptyset \\ \Leftrightarrow & \text{[definition of } \sim \text{]} \\ & \mathbf{a}P = \mathbf{a}({\sim}R) \ \wedge\ \mathbf{b}P \cap \mathbf{b}({\sim}R) = \emptyset \ \wedge\ \mathbf{t}P \cap \mathbf{t}({\sim}R) = \emptyset \\ \Leftrightarrow & \text{[definition of } \| \text{]} \\ & P \rceil\lceil {\sim}R = \langle \emptyset, \emptyset, \emptyset \rangle \end{array}$$

□

Property 5.5

$$P \rceil\lceil R \subseteq S \Leftrightarrow R \subseteq F(P,S)$$

proof:

$$\begin{array}{ll} & P \,\|\, R \subseteq S \\ \Leftrightarrow & \\ & R \rceil\lceil P \subseteq S \\ \Leftrightarrow & \text{[property 5.4 and definition of } \subseteq \text{]} \\ & (R \rceil\lceil P) \rceil\lceil {\sim}S = \mathbf{empty}(\emptyset) \ \wedge\ \mathbf{a}P \div \mathbf{a}R = \mathbf{a}S \\ \Leftrightarrow & \text{[} \mathbf{a}P \cap \mathbf{a}R \cap \mathbf{a}S = \emptyset \Rightarrow \| \text{ is associative]} \\ & R \,\|\, (P \,\|\, {\sim}S) = \mathbf{empty}(\emptyset) \ \wedge\ \mathbf{a}P \div \mathbf{a}R = \mathbf{a}S \\ \Leftrightarrow & \text{[property 1.8 (b) and set theory]} \\ & R \rceil\lceil {\sim}{\sim}(P \rceil\lceil {\sim}S) = \mathbf{empty}(\emptyset) \ \wedge\ \mathbf{a}R = \mathbf{a}P \div \mathbf{a}S \\ \Leftrightarrow & \text{[property 5.4 and definition of } \subseteq \text{]} \\ & R \subseteq {\sim}(P \rceil\lceil {\sim}S) \end{array}$$

□

Theorem 5.6 *The control problem has a solution if and only if*

$$L_{\min} \subseteq G(P, L_{\max})$$

and, if it is solvable, the greatest solution (with respect to $\subseteq$) is $F(P, L_{\max})$.
proof:

$$\begin{array}{ll} & (\exists R :: L_{\min} \subseteq P \rceil\lceil R \subseteq L_{\max}) \\ \Leftrightarrow & [\text{ property 5.5 }] \\ & (\exists R :: L_{\min} \subseteq P \rceil\lceil R \ \wedge \ R \subseteq F(P, L_{\max})) \\ \Leftrightarrow & [\text{ } \Rightarrow\text{: } \rceil\lceil \text{ is monotonic, } \Leftarrow\text{: take } R = F(P, L_{\max}) \text{ }] \\ & L_{\min} \subseteq P \rceil\lceil F(P, L_{\max}) \end{array}$$

It is straightforward to see that $F(P, L_{\max})$ is indeed the greatest possible solution. □

It is clear that every solution of the control problem satisfies

$$R \subseteq F(P, L_{\max})$$

Each $R \subseteq F(P, L_{\max})$ with $L_{\min} \subseteq P \,||\, R$ is a solution.

Exercise 5.1 Define $E(P, L) = \sim(P \,||\, \sim L)$ and proof (if $\mathbf{a}P = \mathbf{a}R$):

$$P \subseteq R \Leftrightarrow P \,||\, \sim R = \mathbf{empty}(\mathbf{a}P)$$

Next, give conditions for which (with ($\mathbf{a}R \subseteq \mathbf{a}P$ and $\mathbf{a}S \subseteq \mathbf{a}P$) the following holds:

$$P \,||\, R \subseteq S \Leftrightarrow R \subseteq E(P, S)$$

❑

Usually, P, $L_{\min}$, and $L_{\max}$ are realistic and we search for a controller R that is also realistic. Notice that, in general, if P and L are realistic, the system $F(P, L)$ need not be. For example take

$$\begin{array}{lcl} P & = & \langle \{a, b\}, \mathbf{pref}(a^*bb), (ab) \rangle \\ L & = & \langle \{a\}, \epsilon, \epsilon \rangle \end{array}$$

then we have:

$$\begin{array}{lcll} & & \sim L & = & \langle \{a\}, a^+, a^+ \rangle \\ & & P \rceil\lceil \sim L & = & \langle \{b\}, \mathbf{pref}(bb), b \rangle \\ F(P, L) & = & \sim(P \rceil\lceil \sim L) & = & \langle \{b\}, (bbb^+), (\epsilon | bb^+) \rangle \end{array}$$

The behaviour set (bbb^+) is not prefix-closed, nor do we have $(\epsilon | bb^+) \subseteq (bbb^+)$. R does not satisfy $\mathbf{b}R = \mathbf{pref}(\mathbf{b}R)$ nor $\mathbf{t}R \subseteq \mathbf{b}R$, so it is not realistic.

It is straightforward to see that the greatest solution of our control problem that is realistic is the greatest realistic DES that is a subsystem of $F(P, L_{\max})$. This is the realistic interior of $F(P, L_{\max})$.

Theorem 5.7 *The control problem has a DES-solution if and only if*

$$L_{\min} \subseteq P \rceil\lceil \mathbf{real}(F(P, L_{\max}))$$

and, if a DES-solution exists, the greatest one is $\mathbf{real}(F(P, L_{\max}))$.
proof: Directly from property 1.6. □

We denote **real**$(F(P,L))$ by $F_{\text{real}}(P,L)$ and also $G_{\text{real}}(P,L) = P \rceil\lceil F_{\text{real}}(P,L)$.

The algorithm to find the needed controller is called the *deCODEr* and can now be described as follows:

Algorithm 5.8

for all L **such that** $L_{\min} \subseteq L \subseteq L_{\max}$:
if $L_{\min} \subseteq G_{\text{real}}(P,L) \subseteq L_{\max}$
then $F_{\text{real}}(P,L)$ is a solution □

Example 5.9 Consider:

$$P = \langle \{a,b,d,e\}, (ae|ad|be) \rangle \quad L = \langle \{d,e\}, (e) \rangle$$

Take $R = \langle \{a,b\}, (a|b) \rangle$, then we have $R \subseteq P\lfloor \mathbf{c}P$ and $\mathbf{t}(P \parallel R) = (d|e)$ (so: $L \subseteq (P \parallel R)\lfloor \mathbf{e}P$), but $\mathbf{t}(F(P,L)) = (b)$ so $F(P,L) \not\subseteq R$. We conclude that we cannot prove:

$$R \subseteq P\lfloor \mathbf{c}P \ \wedge \ L \subseteq (P \rceil\lceil R) \ \Rightarrow \ F(P,L) \subseteq R$$

and therefore, we cannot find in general a smallest possible solution. In other words, as we shall see, if a solution exists, the solution constructed via $L_{\max}$ is a most *liberal* solution. There may or may not be a most *conservative* one. □

Lemma 5.10

$$L_1 \subseteq L_2 \ \Rightarrow \ F(P,L_1) \subseteq F(P,L_2)$$

proof:

$L_1 \subseteq L_2$
$\Leftrightarrow$ [property 1.8 (a)]
$\sim L_1 \supseteq \sim L_2$
$\Rightarrow$ [property 1.17 (b)]
$P \rceil\lceil \sim L_1 \supseteq P \rceil\lceil \sim L_2$
$\Leftrightarrow$ [property 1.8 (a)]
$\sim(P \rceil\lceil \sim L_1) \subseteq \sim(P \rceil\lceil \sim L_2)$
$\Leftrightarrow$ [definition of F]
$F(P,L_1) \subseteq F(P,L_2)$

□

Lemma 5.11

$$L_1 \subseteq L_2 \ \Rightarrow \ G(P,L_1) \subseteq G(P,L_2)$$

proof:

$L_1 \subseteq L_2$
$\Rightarrow$ [lemma 5.10]
$F(P,L_1) \subseteq F(P,L_2)$
$\Rightarrow$ [property 1.17 (b)]
$P \parallel F(P,L_1) \subseteq P \parallel F(P,L_2)$
$\Leftrightarrow$ [definition of $G(P,L)$]
$G(P,L_1) \subseteq GP,(L_2)$

□

This lemma states that an increasing set of choices of L leads to an increasing set of resulting exogenous tasks $(P \parallel F(P,L))\lceil \mathbf{e}P$ of the CODE problem (monotonicity).

Exercise 5.2 Solve the control problem of example 5.2, with $\mathbf{c}P = \{c, b, d\}$. Use state graphs for the computation. □

5.4 Some properties of the deCODEr

In this section we give some properties of solutions of CODE, as constructed using the friend and the guardian.

Property 5.12

(a) $F(P,L_1) \cup F(P,L_2) \subseteq F(P, L_1 \cup L_2)$
(b) $F(P,L_1) \cap F(P,L_2) = F(P, L_1 \cap L_2)$

proof: Part (a):

$$
\begin{array}{ll}
 & F(P,L_1) \cup F(P,L_2) \\
= & \quad [\text{ definition of } F\ ] \\
 & \sim(P \rceil\lceil \sim L_1) \cup \sim(P \rceil\lceil \sim L_2) \\
= & \quad [\text{ property 1.24 (b) }] \\
 & \sim((P \rceil\lceil \sim L_1) \cap (P \rceil\lceil \sim L_2)) \\
\subseteq & \quad [\text{ property 1.23 (e) and 1.8 (a) }] \\
 & \sim(P \parallel (\sim L_1 \cap \sim L_2)) \\
= & \quad [\text{ property 1.24 (a) }] \\
 & \sim(P \rceil\lceil \sim(L_1 \cup L_2)) \\
= & \quad [\text{ definition of } F\ ] \\
 & F(P, L_1 \cup L_2)
\end{array}
$$

Part (b) is similar. □

Lemma 5.13 *If R_1 and R_2 are solutions for CODE then also $R_1 \cup R_2$ is a solution.*
proof:

$$
\begin{array}{ll}
 & L_{\min} \subseteq P \rceil\lceil R_1 \subseteq L_{\max} \ \wedge\ L_{\min} \subseteq P \rceil\lceil R_2 \subseteq L_{\max} \\
\Leftrightarrow & \\
 & L_{\min} \subseteq (P \rceil\lceil R_1) \cup (P \rceil\lceil R_2) \subseteq L_{\max} \\
\Leftrightarrow & \quad [\text{ property 1.23 (a) }] \\
 & L_{\min} \subseteq P \parallel (R_1 \cup R_2) \subseteq L_{\max}
\end{array}
$$

□

This lemma implies that a greatest solution (contained in $P\lceil \mathbf{c}P$) exists, which we already know from theorem 5.7.

In general, however, $R_1 \cap R_2$ and $R_1 \parallel R_2$ need not be solutions, which prevents the existence of a minimal solution, as is shown in the following exercise.

Exercise 5.3 Consider

$$P = \langle\{a,b,c,e\},(ebc|bea)\rangle \quad R_1 = \langle\{a,b,c\},(bc)\rangle \quad L_{\min} = L_{\max} - \langle\{e\},(e)\rangle$$
$$\mathbf{e}P = \{e\} \qquad\qquad R_2 = \langle\{a,b,c\},(ba)\rangle$$

Show that R_1 and R_2 are solutions, but $R_1 \cap R_2$ is not. □

Property 5.14

(a) $R \subseteq F(P, P \rceil\!\lceil R)$
(b) $G(P, P \,||\, R) = P \,||\, R$

proof: Part (a) follows immediately from property 5.5 if we take $S = P \,||\, R$. For part (b) we prove that $G(P, P \rceil\!\lceil R) \subseteq P \rceil\!\lceil R$ because $\supseteq$ follows from (a). We show that the task set of the lhs. is included in the rhs.

$$\begin{array}{ll}
& x \in \mathbf{t}G(P, P \rceil\!\lceil R) \\
\Leftrightarrow & [\text{ definition of } G\ ] \\
& x \in \mathbf{t}(P \,||\, F(P \,||\, R)) \\
\Leftrightarrow & [\text{ definition of } \rceil\!\lceil\ ] \\
& (\exists y : y \in \mathbf{t}P \ \wedge\ y\lceil \mathbf{c}P \in \mathbf{t}(F(P, P \rceil\!\lceil R)) : y\lceil \mathbf{e}P = x) \\
\Leftrightarrow & [\text{ definition of } F \text{ and } \sim \text{ and } y \in \mathbf{t}P \Rightarrow y|\mathbf{c}P \in (\mathbf{e}P)^*\ ] \\
& (\exists y : y \in \mathbf{t}P \ \wedge\ y\lceil \mathbf{c}P \notin \mathbf{t}(P \rceil\!\lceil \sim(P \rceil\!\lceil R)) : y\lceil \mathbf{e}P = x) \\
\Leftrightarrow & [\text{ definition of } \rceil\!\lceil\ ] \\
& (\exists y : y \in \mathbf{t}P \ \wedge\ (\forall z : z \in \mathbf{t}P \ \wedge\ z\lceil \mathbf{e}P \in \mathbf{t}(\sim(P \rceil\!\lceil R)) : z\lceil \mathbf{c}P \neq y\lceil \mathbf{c}P) : y\lceil \mathbf{e}P = x) \\
\Leftrightarrow & [\text{ definition of } \sim \text{ and } z \in \mathbf{t}P \Rightarrow z\lceil \mathbf{e}P \in (\mathbf{e}P)^*\ ] \\
& (\exists y : y \in \mathbf{t}P \ \wedge\ (\forall z : z \in \mathbf{t}P \ \wedge\ z\lceil \mathbf{e}P \notin \mathbf{t}(P \rceil\!\lceil R) : z\lceil \mathbf{c}P \neq y\lceil \mathbf{c}P) : y\lceil \mathbf{e}P = x) \\
\Leftrightarrow & [\text{ definition of } ||\ ] \\
& (\exists y : y \in \mathbf{t}P \ \wedge\ (\forall z : z \in \mathbf{t}P \ \wedge\ (\forall u : u \in \mathbf{t}P \ \wedge\ u\lceil \mathbf{c}P \in \mathbf{t}R : u\lceil \mathbf{e}P \neq z\lceil \mathbf{e}P) \\
& \qquad\qquad : z\lceil \mathbf{c}P \neq y\lceil \mathbf{c}P) \\
& \quad : y\lceil \mathbf{e}P = x) \\
\Leftrightarrow & [\text{ trading }] \\
& (\exists y : y \in \mathbf{t}P \ \wedge\ (\forall z : z \in \mathbf{t}P \ \wedge\ z\lceil \mathbf{c}P = y\lceil \mathbf{c}P \\
& \qquad\qquad : (\exists u : u \in \mathbf{t}P \ \wedge\ u\lceil \mathbf{c}P \in \mathbf{t}R : u\lceil \mathbf{e}P = z\lceil \mathbf{e}P)) \\
& \quad : y\lceil \mathbf{e}P = x) \\
\Rightarrow & [\text{ take } z = y\ ] \\
& (\exists y : y \in \mathbf{t}P \ \wedge\ (\exists u : u \in \mathbf{t}P \ \wedge\ u\lceil \mathbf{c}P \in \mathbf{t}R : u\lceil \mathbf{e}P = y\lceil \mathbf{e}P)) : y\lceil \mathbf{e}P = x) \\
\Rightarrow & \\
& (\exists u : u \in \mathbf{t}P \ \wedge\ u\lceil \mathbf{c}P \in \mathbf{t}R : u\lceil \mathbf{e}P = x) \\
\Leftrightarrow & \\
& x \in \mathbf{t}(P \rceil\!\lceil R)
\end{array}$$

Similar for the behaviour set, which completes the proof. □

Property 5.15

(a) $F(P_1 \cup P_2, L) = F(P_1, L) \cup F(P_2, L)$
(b) $F(P_1 \cap P_2, L) \supseteq F(P_1, L) \cap F(P_2, L)$

proof:

$$
\begin{aligned}
& F(P_1 \cup P_2, L) \\
= \quad & [\text{ definition of } F \text{ }] \\
& \sim((P_1 \cup P_2) \rceil\!\lceil \sim L) \\
= \quad & [\text{ property 1.23 (d) }] \\
& \sim((P_1 \rceil\!\lceil L) \cup (P_2 \rceil\!\lceil L)) \\
= \quad & [\text{ property 1.24 }] \\
& (\sim(P_1 \rceil\!\lceil L)) \cup \sim(P_2 \rceil\!\lceil \sim L)) \\
= \quad & [\text{ definition of } F \text{ }] \\
& F(P_1, L) \cup F(P_2, L)
\end{aligned}
$$

Part (b) is similar (no equality because of property 1.23 (e). □

5.5 Observability

Next, we would like to investigate whether every solution of CODE can be written in terms of a friend of some L satisfying the minmax condition. Because every solution of CODE can be extended with tasks that have no influence on the result (take $R_{\text{new}} = R \cup R_{\text{ext}}$ for an R_{ext} with $P \parallel R_{\text{ext}} = \mathbf{empty}(\mathbf{a}P \cup \mathbf{a}R)$, then R_{new} is also a solution), and also, from every solution such tasks can be deleted, we can only wish that every solution R with $R \subseteq P\lceil\mathbf{c}P$ can be written in terms of a certain friend. Therefore, we introduce the notion *effective part* of a solution R, which is equal to

$$R \cap P\lceil\mathbf{c}P$$

Now suppose R is a solution of CODE, then $P \rceil\!\lceil R$ satisfies the minmax condition. From property 5.14 (a) we have:

$$R \subseteq F(P, P \rceil\!\lceil R)$$

In general, we have no equality here, as is shown in the following example.

Example 5.16 Consider

$$
\begin{aligned}
& P = \langle\{a, b, d, e\}, (adbe|adae)\rangle \\
& \mathbf{e}P = \{d, e\} \\
& L_{\min} = L_{\max} = \langle\{d, e\}, (de)\rangle
\end{aligned}
$$

then $x_1 = adbe$ leads to $x_1\lceil\mathbf{c}P = ab$ and $x_1\lceil\mathbf{e}P = de$ and $x_2 = adae$ leads to $x_2\lceil\mathbf{c}P = aa$ and $x_2\lceil\mathbf{e}P = de$. We know that $R = \langle\{a, b\}, (aa)\rangle$ is a solution of CODE, because $\mathbf{t}(P \parallel R)\lceil\mathbf{e}P = (de)$ and $R \subseteq P\lceil\mathbf{c}P$, but no L satisfying the minmax condition can be found so that $F(P, L) \cap P\lceil\mathbf{c}P = R$. The only possible L, namely $L = L_{\min}$, leads to $F(P, L) \cap P\lceil\mathbf{c}P = \langle\{a, b\}, (ab|aa)\rangle \neq R$.

In this case there are more communication tasks $y \in \mathbf{t}P\lceil\mathbf{c}P$ that lead to the same exogenous task ($y = aa$ as well as $y = ab$ leads to the exogenous task de). To find a solution not all these communication tasks are needed. Just one will do, but using the

deCODEr all possible communication tasks are given. □

It is easily seen that if in P all exogenous tasks can be found by applying a unique communication task only, all solutions of CODE can be found by applying the deCODEr (i.e., are of the form $F(P, L)$). A system P with this property is called observable.

Definition 5.17 *P is observable with respect to some exogenous alphabet E, notation* **observable**$_E(P)$*, if (with $C = \mathbf{a}P \backslash E$)*

$$(\forall x, y : x \in \mathbf{t}P \ \wedge \ y \in \mathbf{t}P : x \lceil E = y \lceil E \Rightarrow x \lceil C = y \lceil C)$$

□

Exercise 5.4 Which of the following systems are observable, and which of the observable systems has a solution for CODE with $L_{\min} = L_{\max} = \langle \{d, e\}, (e) \rangle$:

(a) $\langle \{a, b, d, e\}, (ae|ad|be) \rangle$
(b) $\langle \{a, b, d, e\}, (ad|be) \rangle$
(c) $\langle \{a, b, d, e\}, (ae|ad) \rangle$

□

Lemma 5.18

observable$_{\mathbf{e}P}(P)$
$\Rightarrow$
$(\forall R : \ R \text{ is a solution of } CODE \ : F(P, P \,]\![\, R) \cap P \lceil \mathbf{c}P = R \cap P \lceil \mathbf{c}P)$

proof: We only have to prove that $F(P, P \,]\![\, R) \cap P \lceil \mathbf{c}P \subseteq R \cap P \lceil \mathbf{c}P$, because the other inclusion immediately follows from property 5.5. We prove that the task set of the first is included in the second system:

$x \in \mathbf{t}(F(P, P \,]\![\, P) \cap P \lceil \mathbf{c}P)$
$\Leftrightarrow$ [definition of $\cap$]
$x \in \mathbf{t}P \lceil \mathbf{c}P \ \wedge \ x \in \mathbf{t}(F(P, P \,]\![\, R)$
$\Leftrightarrow$ [definition of $F(P, L)$]
$x \in \mathbf{t}P \lceil \mathbf{c}P \ \wedge \ x \in (\mathbf{c}P)^* \backslash \mathbf{t}(P \,]\![\, \sim(P \,]\![\, R))$
$\Leftrightarrow$ [$x \in \mathbf{t}P \lceil \mathbf{c}P \Rightarrow x \in (\mathbf{c}P)^*$]
$x \in \mathbf{t}P \lceil \mathbf{c}P \ \wedge \ x \notin \mathbf{t}(P \,]\![\, \sim(P \,]\![\, R))$
$\Leftrightarrow$ [definition of $]\![$]
$x \in \mathbf{t}P \lceil \mathbf{c}P \ \wedge \ (\forall y : y \in \mathbf{t}P \ \wedge \ y \lceil \mathbf{e}P \in \mathbf{t}(\sim(P \,]\![\, R)) : y \lceil \mathbf{c}P \neq x)$
$\Leftrightarrow$ [definition of $\sim$]
$x \in \mathbf{t}P \lceil \mathbf{c}P \ \wedge \ (\forall y : y \in \mathbf{t}P \ \wedge \ y \lceil \mathbf{e}P \in (\mathbf{e}P)^* \backslash \mathbf{t}(P \,]\![\, R) : y \lceil \mathbf{c}P \neq x)$
$\Leftrightarrow$ [$y \in \mathbf{t}P \Rightarrow y \lceil \mathbf{e}P \in (\mathbf{e}P)^*$]
$x \in \mathbf{t}P \lceil \mathbf{c}P \ \wedge \ (\forall y : y \in \mathbf{t}P \ \wedge \ y \lceil \mathbf{e}P \notin \mathbf{t}(P \,]\![\, R) : y \lceil \mathbf{c}P \neq x)$
$\Leftrightarrow$
$x \in \mathbf{t}P \lceil \mathbf{c}P \ \wedge \ (\forall y : y \in \mathbf{t}P \ \wedge \ y \lceil \mathbf{c}P = x : y \lceil \mathbf{e}P \in \mathbf{t}(P \,]\![\, R))$
$\Leftrightarrow$ [definition of $]\![$]
$x \in \mathbf{t}P \lceil \mathbf{c}P \ \wedge \ (\forall y : y \in \mathbf{t}P \ \wedge \ y \lceil \mathbf{c}P = x$
$\qquad : (\exists z : z \in \mathbf{t}P \ \wedge \ z \lceil \mathbf{c}P \in \mathbf{t}R : z \lceil \mathbf{e}P = y \lceil \mathbf{e}P))$
$\Leftrightarrow$

$$
\begin{array}{l}
\quad x \in \mathbf{t}P\lceil\mathbf{c}P \ \wedge\ (\forall y : y \in \mathbf{t}P \ \wedge\ y\lceil\mathbf{c}P = x \\
\qquad\qquad\qquad\qquad : (\exists z : z\lceil\mathbf{e}P = y\lceil\mathbf{e}P \ \wedge\ z \in \mathbf{t}P : z\lceil\mathbf{c}P \in \mathbf{t}R)) \\
\Rightarrow \quad [\ z\lceil\mathbf{e}P = y\lceil\mathbf{e}P \Rightarrow z\lceil\mathbf{c}P = y\lceil\mathbf{c}P,\ P \text{ is observable }] \\
\quad x \in \mathbf{t}P|\mathbf{c}P \ \wedge\ (\forall y : y \in \mathbf{t}P \ \wedge\ y|\mathbf{c}P = x : y|\mathbf{c}P \in \mathbf{t}R) \\
\Rightarrow \\
\quad x \in \mathbf{t}P\lceil\mathbf{c}P \ \wedge\ x \in \mathbf{t}R \\
\Leftrightarrow \\
\quad x \in \mathbf{t}(R \cap P\lceil\mathbf{c}P)
\end{array}
$$

The same can be proven for the behaviour set. □

5.5.1 Relation to conventional system theory

In [Wil88] observability on a dynamical system $\Sigma = (T, W, B)$, with $W = W_1 \times W_2$ and $B \subseteq B_1 \times B_2$, with $B_i = P_{W_i}(B)$ the projection on W_i, is defined by:

> $w_2 : T \to W_2$ is observable from $w_1 : T \to W_1$ if there exists a function $F : B_1 \to B_2$ with $((w_1, w_2) \in B) \Leftrightarrow (w_2 = F(w_1))$.

Translating this to our definition of discrete systems leads to:

> $z \in P\lceil\mathbf{e}P$ is observable from $y \in P\lceil\mathbf{c}P$ if there is some function $F : P\lceil\mathbf{e}P \to P\lceil\mathbf{c}P$ with $(\exists x : x \in \mathbf{t}P : x\lceil\mathbf{c}P = z \ \wedge\ x\lceil\mathbf{e}P = y) \Leftrightarrow (z = F(y))$

If we define F as

$$F(y) = \{x : x \in \mathbf{t}P \ \wedge\ x\lceil\mathbf{e}P = y : x\lceil\mathbf{c}P\}$$

we see that F is a function if and only if P is observable according to our definition.

We conclude that our definition of observability meets the general meaning of the notion in system theory.

5.5.2 CODE for observable systems

From lemma 5.18 it is clear that every effective part of a solution of CODE for an observable discrete system P is equal to the effective part of a solution of the form $F(P, L)$. In that case, the deCODEr gives in essence all possible solutions.

From lemma 5.10 we see that

$$
\begin{array}{l}
\quad L_{\min} \subseteq L \\
\Rightarrow \\
\quad F(P, L_{\min}) \subseteq F(P, L)
\end{array}
$$

holds for every L satisfying the minmax condition. If P is observable, all effective parts of the solutions are of the form $F(P, L) \cap P\lceil\mathbf{c}P$. We conclude that for an observable P the effective part of the solution $F(P, L_{\min})$ is minimal:

Theorem 5.19 *If CODE has a solution, then $F(P, L_{\max}) \cap P\lceil\mathbf{c}P$ is the largest solution contained in $P\lceil\mathbf{c}P$. If in addition P is observable, then $F(P, L_{\min}) \cap P\lceil\mathbf{c}P$ is the least solution contained in $P\lceil\mathbf{c}P$.*

proof: It is clear that $F(P, L_{\max}) \cap P\lceil\mathbf{c}P$ is the largest solution that is contained in $P\lceil\mathbf{c}P$. Suppose R is some solution of CODE and $R \subseteq P\lceil\mathbf{c}P$, then we have

$$
\begin{array}{ll}
 & F(P, L_{\min}) \cap P\lceil \mathbf{c}P \\
\subseteq & [\ R \text{ is a solution, so } L_{\min} \subseteq P \parallel R\ ] \\
 & F(P, P \rceil\!\lceil R) \cap P\lceil \mathbf{c}P \\
= & [\ P \text{ is observable }] \\
 & R \cap P\lceil \mathbf{c}P \\
= & [\ R \subseteq P\lceil \mathbf{c}P\ ] \\
 & R
\end{array}
$$

so $F(P, L_{\min}) \cap P\lceil \mathbf{c}P$ is the smallest solution □

Example 5.20 Reconsider the example of the farmer, wolf, goat, and cabbage of chapter 2 (see figure 2.4). The Dutch puzzle can easily be solved now using the de-CODEr. The question is how should the farmer bring wolf, goat, and cabbage from this side of the river to the other. First, notice that all events are communication events except event e which is an exogenous event (no-one can influence it). Our uncontrolled exogenous behaviour equals ϵ. We do not want e to occur so we want an empty exogenous behaviour, i.e.,

$$L_{\min} = L_{\max} = \langle \{e\}, \epsilon \rangle$$

Solving the problem means computing $F(P, L)$ with P the system as described in figure 2.4 and $L = \langle \{e\}, \epsilon \rangle$ and checking if indeed $L_{\min} \subseteq G(P, L)$. Doing the computation leads to $F(P, L)$ as is displayed in figure 5.1. If we omit unnecessary loops we find only two paths leading from the initial to the (only) final state in this diagram, namely:

gfwgcfg
gfcgwfg

These are indeed known to be the only solutions to the problem. (To convince everyone we may check whether $L_{\min} \subseteq G(P, L_{\max})$. Indeed, this inclusion holds and, in fact, is an equivalence.) □

5.6 Lock free control

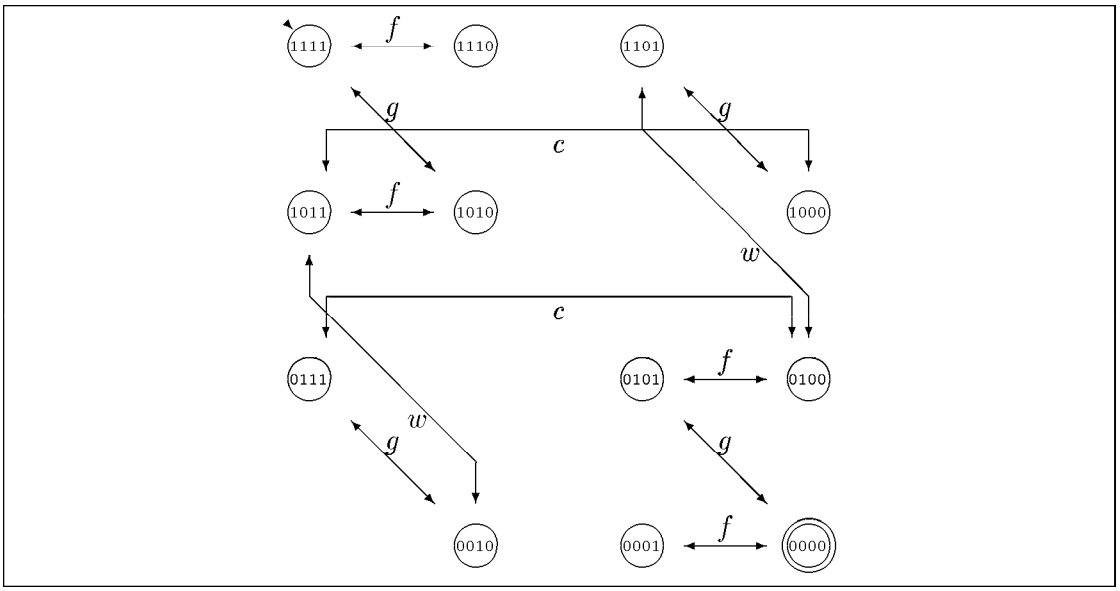
An additional problem arises if one tries to control a system and the resulting controller leads to a connection that is not free of lock. The following example illustrates this.[6]

Consider a house in which a cat and a mouse are living. The house consists of 5 rooms and a number of doors. Cat and mouse have seperate doors to use. Most doors are one-way only. The configuration is given in figure 5.2. The mouse moves are given by the events $m\ n\ o\ p\ q\ r$ and the cat moves by the events $c\ d\ e\ f\ g\ h\ i$.

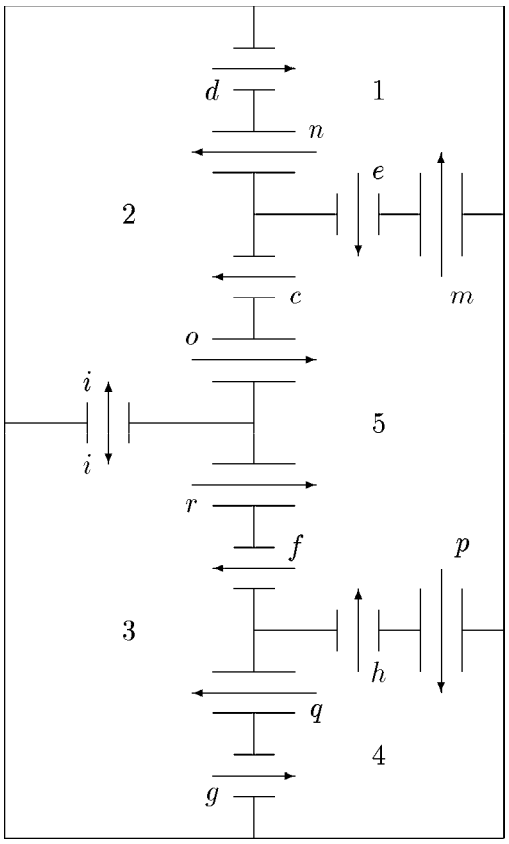
It is not difficult to find a model in state space for this system. Transitions in the modelling graph are (a, b, a', and b' represent one of $1, \ldots, 5$):

$(aa) \xrightarrow{x} \dagger$	cat eats mouse
$(ab) \xrightarrow{y} (a'b)$	cat goes from a to a' via door y
$(ab) \xrightarrow{y} (ab')$	mouse goes from b to b' via door y

[6] Taken from [WR87].



Figuur 5.1: Solution for the farmer problem



Figuur 5.2: Configuration for cat and mouse

Initialy the cat is in room 1 and the mouse is in room 4. So (14) is the initial state. We let cat and mouse freely walk through the house but only have completed behaviour (tasks) if cat and mouse have returned to their initial possitions. Also if the cat has eaten the mouse we say we have a completed task. So the states denoted by (14) and † are task states. In figure 5.3 all transitions are given.

Our control problem now is finding some controller that forbids cat and mouse to go through some doors if this leads to the mouse being eaten by the cat. In our framework this means that we are looking for a controller that allows in each situation those events that do no lead to disaster. The events that are used for control are all events but x. Event x is the only exogenous event that cannot be controlled. We choose:

$$L_{\min} = L_{\max} = \langle \{x\}, \epsilon \rangle$$

i.e., event x does not occur. Computing $F_{\text{des}}(P, L_{\max})$ leads to $P \parallel F_{\text{des}}(P, L_{\max})$ as is given is figure 5.4.

Now suppose also event i is uncontrollable. So we have

$$L_{\max} = L_{\min} = \langle \{x, i\}, i^* \rangle$$

(i may occur, x not).

Again computing $F_{\text{des}}(P, L_{\max})$ leads to a connection that may lock. The connection is given in figure 5.5.

Now it turns out that, although we indeed have established the minmax condition, the connection may lock.

So we should find another controller so that the minmax condition is fulfilled and also the connection is free of lock.

5.6.1 Lock free controller

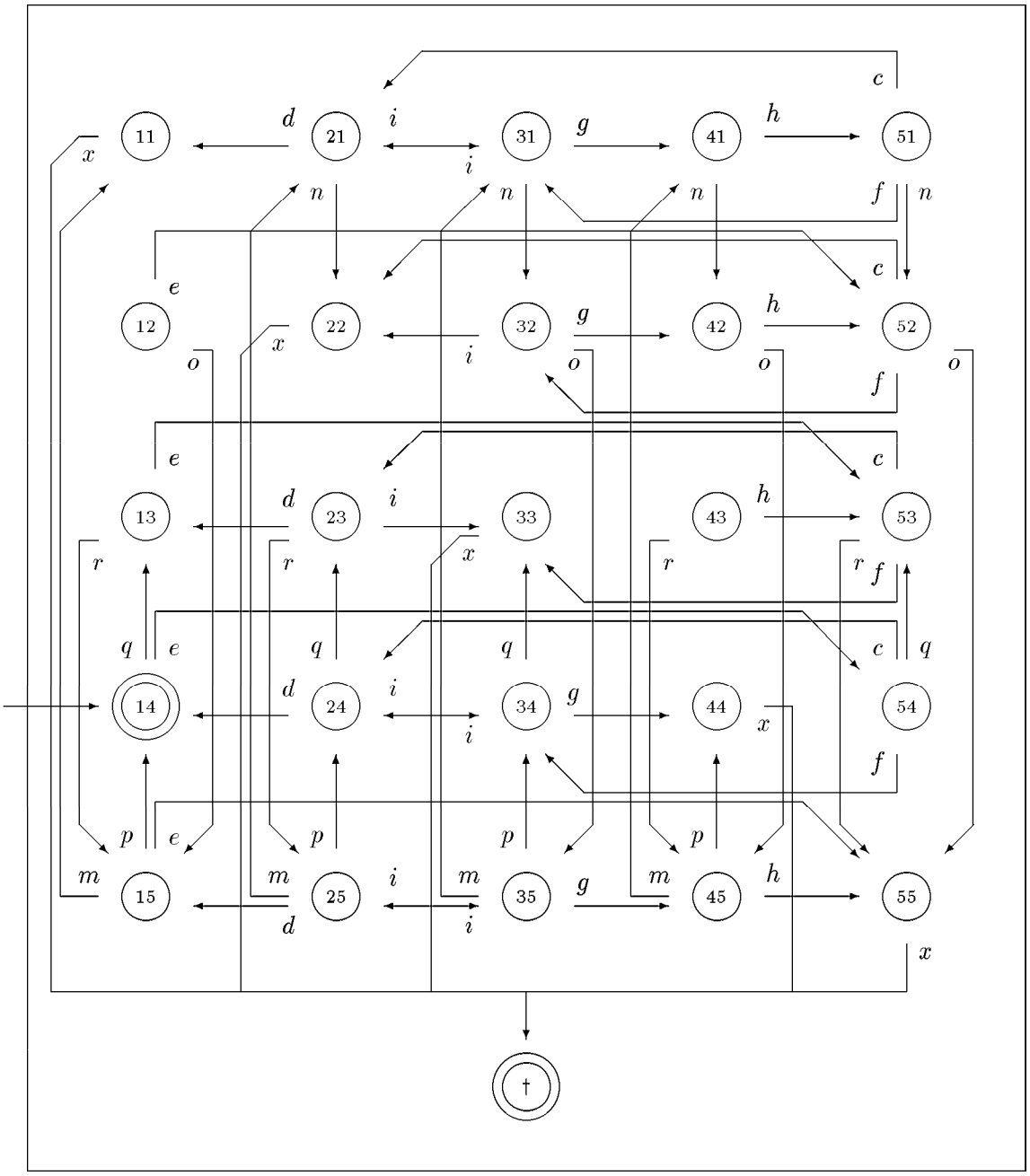
It will not come to a suprise that we combine the methods found in chapter 3 and this chapter to create a controller that leads to a lock free connection.

Theorem 5.21 *CODE is solvable leading to a lock free connection, if and only if*

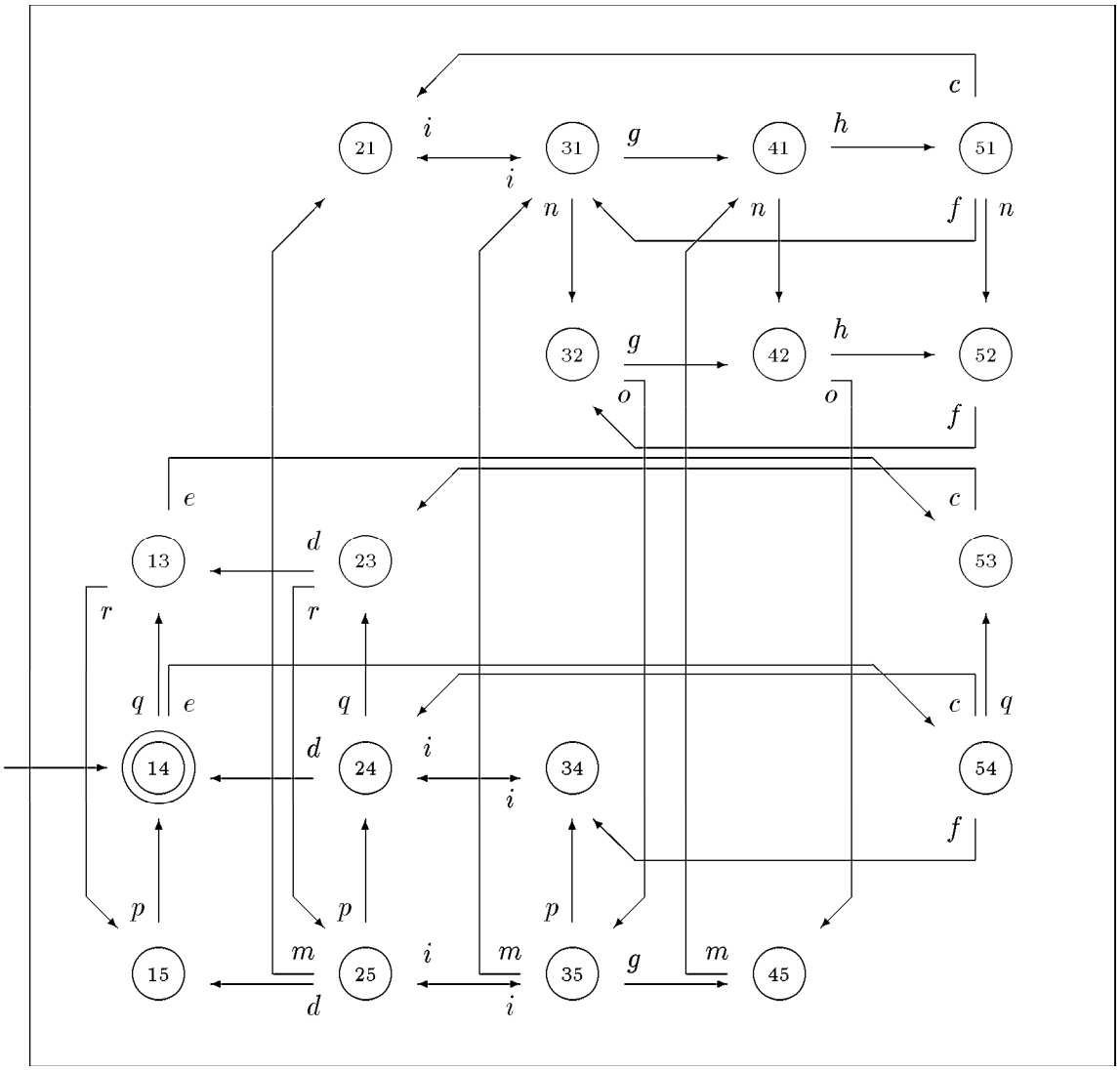
$$L_{\min} \subseteq P \rceil\lceil \Lambda(P, F_{\text{des}}(P, L_{\max}))$$

If this condition is fulfilled, the greatest controller is $\Lambda(P, F_{\text{des}}(P, L_{\max}))$.
proof: The proof follows directly using the fact that both $F(P, L)$ and $\Lambda(P, L)$ return the greatest subsystems. □

We now can apply this theory to our example. In figure 5.6 the resulting connection is given, which indeed is free of lock and has the desired behaviour.



Figuur 5.3: State graph for cat and mouse



Figuur 5.4: Controlled cat and mouse system

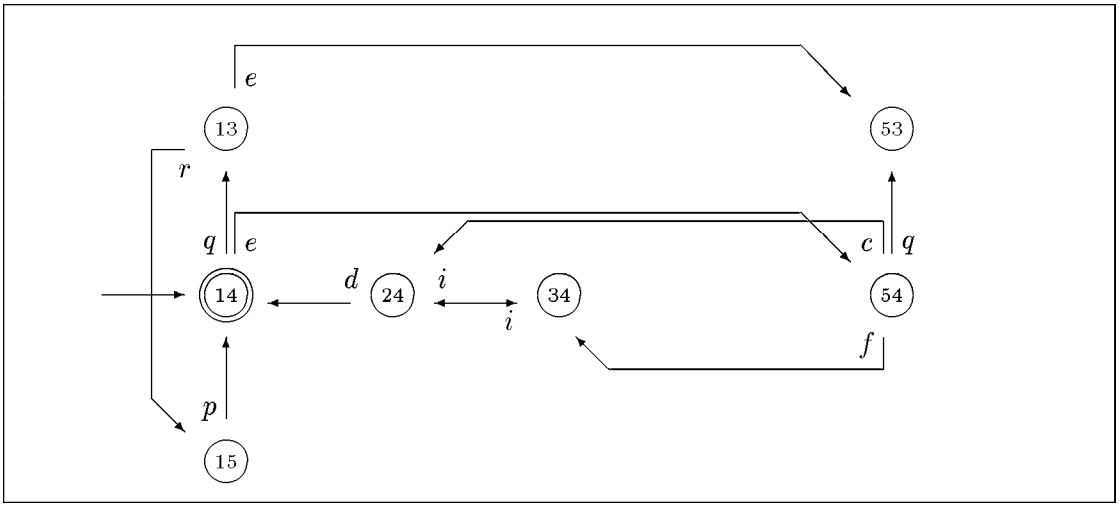
5.7 A more general setting for CODE

So far, we have only considered CODE in the special case in which exactly all communications of P are used to control exactly all exogenous events of P. However, it is possible to define CODE in a more general setting:

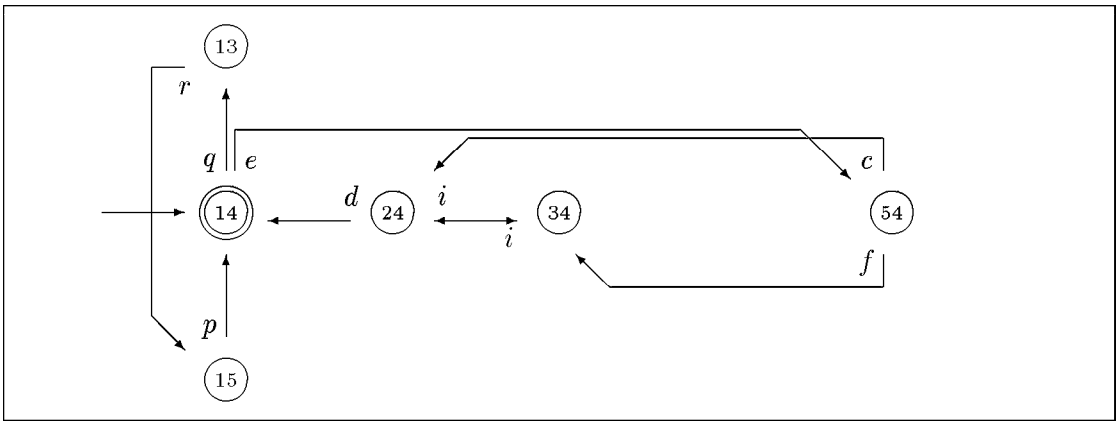
1) the controller R may also have exogenous events,[7]
2) R may have a set of communication events that is larger than P,
3) R may have a set of communication events that is smaller than P,
4) $L_{\min}$ and $L_{\max}$ may be subsets of only part of the exogenous behaviour of P.

The first two cases are of no interest: Extra exogenous or communication events in R do not contribute to the control of the exogenous events of P. The last two cases, however,

[7] These events do not appear in $\mathbf{e}P$.



Figuur 5.5: Second controlled cat and mouse system. State 53 is a deadlock state



Figuur 5.6: Second lock free controlled cat and mouse system

are of interest. If R has a smaller set of communication events than P, we have to try to control the exogenous behaviour of P using only part of the communication events. If $L_{\min}$ and $L_{\max}$ are subsets of a partial exogenous behaviour of P, we only have to control a subset of all exogenous events of P.

We extend the formulation of CODE in such a way that these possibilities are included also. We find the following more general setting of CODE:
Given

$P = \langle \mathbf{a}P\mathbf{b}P, , \mathbf{t}P \rangle$
$E \subseteq \mathbf{e}P$
$L_{\min} \subseteq L_{\max}$

Find

$R = \langle \mathbf{a}R, \mathbf{b}R, \mathbf{t}R \rangle$ with $\mathbf{a}R \subseteq \mathbf{c}P$

Such that

$L_{\min} \subseteq (P \parallel R)\lceil E \subseteq L_{\max}$

It is possible (as will be shown) to transform this extended CODE problem to the original one, find a solution, and retransform this solution to become a solution of the extended CODE problem.

We first deal with the case that $\mathbf{a}R \subset \mathbf{c}P$. If we may use only fewer communication events than are present in P, we (temporarily) use exogenous alphabet $\mathbf{e}P' = \mathbf{a}P \setminus \mathbf{a}R$ and $\mathbf{c}P' = \mathbf{a}R$. communication events are considered as exogenous events: We have enlarged the set of exogenous events and have to solve CODE in the case that not all exogenous events have to be controlled.

It turns out that it is satisfactory to consider only the case in which not all exogenous events have to be controlled and $\mathbf{c}P = \mathbf{a}R$ (if not so, first do the transformation as prescribed above).
So consider:

$P = \langle \mathbf{a}P, \mathbf{b}P, \mathbf{t}P \rangle$
$L_{\min} \subseteq L_{\max}$
$E \subset \mathbf{e}P$

and try to find $R = \langle \mathbf{c}P, \mathbf{b}R, \mathbf{t}R \rangle$ such that:

$$L_{\min} \subseteq (P \parallel R)\lceil E \subseteq L_{\max}$$

To find a solution for this problem, we extend $L_{\min}$ and $L_{\max}$ to become subsets of $P\lceil \mathbf{e}P$ as follows:

$L^e_{\min} = L_{\min} \parallel P\lceil \mathbf{e}P$
$L^e_{\max} = L_{\max} \parallel P\lceil \mathbf{e}P$

We have arranged that $L^e_{\min}$ and $L^e_{\max}$ are subsets of $P\lceil \mathbf{e}P$ and that the desired behaviour of events from E is still prescribed by $L^e_{\min}$ and $L^e_{\max}$. The CODE problem so arranged is denoted by CODE^e.

We will prove that finding a solution for CODE^e leads directly to the solution for the original CODE-problem, i.e., if $R = \langle \mathbf{t}R, \mathbf{c}P \rangle$ is a solution for CODE^e, then it is a solution for CODE.

Therefore, it remains to prove that (with $L^e_{\min} \subseteq L \subseteq L^e_{\max}$):

$L^e_{\min} \subseteq G(P, L) \subseteq L^e_{\max}$
$\Rightarrow$
$L_{\min} \subseteq G(P, L)\lceil E \subseteq L_{\max}$

First notice:

$L^e_{\min}\lceil E$
$=$ [definition of $L^e_{\min}$]
$(L_{\min} \parallel P\lceil \mathbf{e}P)\lceil E$
$=$ [property 1.21 (b), note: $\mathbf{a}L_{\min} \cap \mathbf{a}(P\lceil \mathbf{e}P) = E \subseteq E$]
$L_{\min}\lceil E \parallel P\lceil \mathbf{e}P\lceil E$
$=$ [$P\lceil \mathbf{e}P\lceil E = P\lceil E$ and property 1.20 (a)]
$L_{\min} \cap P\lceil E$
$=$ [$L_{\min} \subseteq P\lceil E$]
$L_{\min}$

and, similarly:

$$L^e_{\max} \lceil E = L_{\max}$$

We have:

$$\begin{array}{ll} & L^e_{\min} \subseteq G(P,L) \subseteq L^e_{\max} \\ \Rightarrow & [\ \lceil \text{ is monotonic }] \\ & L^e_{\min} \lceil E \subseteq G(P,L) \lceil E \subseteq L^e_{\max} \lceil E \\ \Leftrightarrow & \\ & L_{\min} \subseteq G(P,L) \lceil E \subseteq L_{\max} \end{array}$$

So, in general, each solution of CODE^e is a solution of CODE.

It turns out that control of only part of the exogenous events means that we do not give any constraint for the uncontrolled exogenous events. All possibilities of occurrence of this uncontrolled behaviour (as is possible in P) are simply copied to the constraints $L_{\min}$ and $L_{\max}$ by means of the shuffle operator.

Exercise 5.5 Reconsider example 5.2:

$$\begin{array}{lcl} P & = & \langle \{a,b,c,d,e,g\}, \mathbf{pref}(ae|ad|ag|be|cg), (ae|be|cg\rangle \\ \mathbf{e}P & = & \{d,e,g\} \\ \mathbf{t}L_{\min} & = & e \\ \mathbf{t}L_{\max} & = & (e|g) \\ \mathbf{c}P & = & \{a,b\} \end{array}$$

Find a controller R with $\mathbf{a}R = \mathbf{c}P$ to solve this CODE problem. □

5.8 The extended control problem

Instead of looking at $L_{\min} \subseteq P \rceil\lceil R \subseteq L_{\max}$, one might consider looking at

$$L_{\min} \subseteq P \parallel R \subseteq L_{\max}$$

which has some similarity in supervisory control theory of Wonham and Ramadge (see [RW87]). First, we give a definition and solution for this new problem.

In this section we discuss the following problem:
Given a discrete process P with $\mathbf{c}P \subseteq \mathbf{a}P$ and two discrete processes $L_{\min}$ and $L_{\max}$ with

$$L_{\min} \subseteq L_{\max} \subseteq \mathbf{chaos}(\mathbf{a}P)$$

Find, if possible, a controller discrete process R with $\mathbf{a}R = \mathbf{c}P$

$$L_{\min} \subseteq P \parallel R \subseteq L_{\max}$$

This problem is called the *extended control problem* or ECODE for short. Notice that the minmax condition restricts the total behaviour here. Without loss of generality we assume that

$$L_{\min} \subseteq L_{\max} \subseteq P$$

(we cannot create traces in $P \parallel R$ that are not in P).

As we shall see, the solution of this problem is very much like the solution of CODE itself: we use the same friend of L, but have to subtract some extra traces from its behaviour to get the right controller.

To solve the problem, we need the following functions on discrete processes.

Definition 5.22 *With the ECODE problem we associate the discrete processes:*

$$E(P,L) = F(P, L\lceil \mathbf{e}P) \cap L\lceil \mathbf{c}P$$

called the extended friend *of L, and*

$$H(P,L) = P \parallel E(P,L)$$

called the host *of L.* □

$F(P, L\lceil \mathbf{e}P)$ is a possible candidate for solving CODE. $E(P,L)$ is a possible candidate for solving ECODE (created by using $F(P, L\lceil \mathbf{e}P)$ and deleting all communicating traces that are not desired in ECODE); $H(P,L)$ is like the guardian in CODE: it gives the resulting behaviour using the extended friend.

Theorem 5.23

ECODE is solvable

$\Leftrightarrow$

$L_{\min} \subseteq H(L_{\max})$

and in case ECODE is solvable a solution is $E(P, L_{\max})$. □

To prove the theorem, we need the following property:

Property 5.24 *For $S \subseteq P$ we have:*

$$P \parallel R \subseteq S \Leftrightarrow R \subseteq E(P,S)$$

proof: (take $C = \mathbf{c}P$ and $B = \mathbf{a}P \backslash C$)

$R \subseteq E(P,S)$

$\Leftrightarrow$ [definiton of E]

$R \subseteq F(P, S\lceil B) \cap S\lceil C$

$\Leftrightarrow$ [property 5.5]

$P \rceil\lceil R \subseteq S\lceil B \ \wedge \ R \subseteq S\lceil C$

$\Leftrightarrow$ [definition of $\rceil\lceil$]

$(P \parallel R)\lceil B \subseteq S\lceil B \ \wedge \ R \subseteq S\lceil C$

$\Leftrightarrow$ [$R \subseteq P\lceil C$]

$(P \parallel R)\lceil B \subseteq S\lceil B \ \wedge \ (P\lceil C) \cap R \subseteq S\lceil C$

$\Leftrightarrow$ [property 1.20]

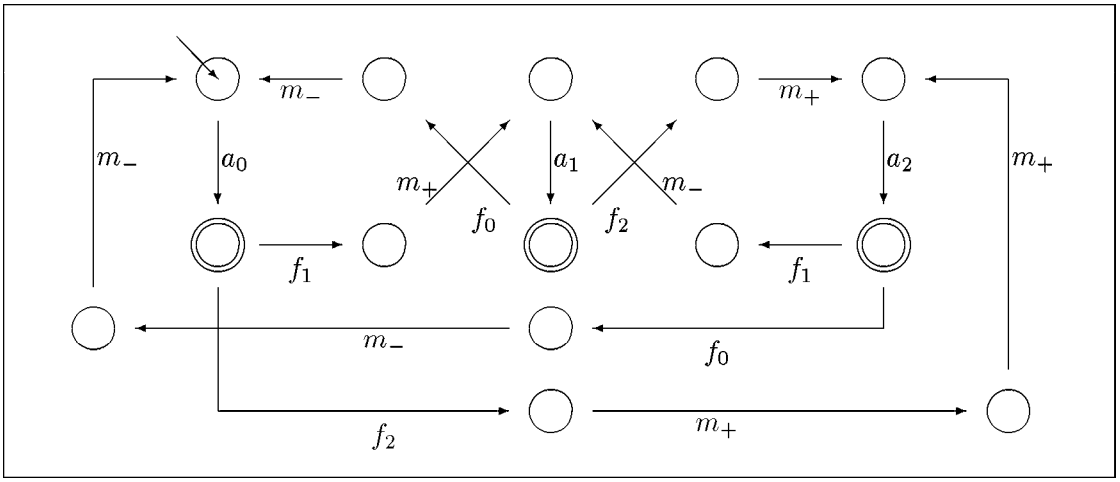
$(P \parallel R)\lceil B \subseteq S\lceil B \ \wedge \ (P\lceil C) \parallel R \subseteq S\lceil C$

$\Leftrightarrow$

$(P \parallel R)\lceil B \subseteq S\lceil B \ \wedge \ (P \parallel R)\lceil C \subseteq S\lceil C$

$\Leftrightarrow$ [$S \subseteq P$]

$P \parallel R \subseteq S$



Figuur 5.7: Behaviour of the elevator

❑

proof: (of theorem 5.23)

$$
\begin{array}{ll}
 & (\exists R : L_{\min} \subseteq P \parallel R \subseteq L_{\max}) \\
\Leftrightarrow & [\text{ property 5.24 }] \\
 & (\exists R : L_{\min} \subseteq P \parallel R \ \wedge \ R \subseteq E(P, L_{\max})) \\
\Leftrightarrow & [\ \rightarrow\text{: } \parallel \text{ is monotonic; } \leftarrow\text{: take } R = E(P < L_{\max})\ ] \\
 & (\exists R : L_{\min} \subseteq P \parallel E(P, L_{\max}))
\end{array}
$$

❑

Example 5.25 Consider the following model of an elevator P, as given in figure 5.7, with events:

m_+	elevator going up one floor
m_-	elevator going down one floor
f_i	elevator needs to go to floor i
a_i	elevator arrives at floor i

We suppose $\mathbf{e}P = \{m_+, m_-\}$ and $\mathbf{c}P = \{a_0, a_1, a_2, f_0, f_1, f_2\}$.

A person want to go from floor 1 to floor 2. Exogenous behaviour m_+m_+, however, is not enough to guarantee that the person can use the elevator. It should stop at floor 1 also. Therefore, we use the ECODE formulation here, i.e., we want to have the desired behaviour

$$\mathbf{t}L = a_0 \cdot f_1 \cdot m_+ \cdot a_1 \cdot f_2 \cdot m_+ \cdot a_2$$

Computations give:

$$
\begin{array}{lcl}
\mathbf{t}L\lceil\mathbf{e}P & = & m_+ \cdot m_+ \\
\mathbf{t}F(P, L\lceil\mathbf{e}P) & = & (a_0 \cdot f_1 \cdot a_1 \cdot f_2 \cdot a_2 | a_0 \cdot f_2 \cdot a_2) \\
\mathbf{t}E(P, L) & = & a_0 \cdot f_1 \cdot a_1 \cdot f_2 \cdot a_2
\end{array}
$$

and indeed $H(P, L) = L$. ❑

Exercise 5.6 Under what conditions is $L_{\min} \subseteq P \parallel R \subseteq L_{\max}$ solvable with $R = \sim(P \parallel \sim L_{\max})$ (recall exercise 5.1). □

5.9 References

Control of discrete event systems is also studied by prof. Wonham and prof. Ramadge in their supervisory control theory. In fact, the minmax condition is adapted from a similar condition from that theory, see [RW87]. The supervisory control theory is discussed in appendix S.

The control problem as formulated in this chapter first appeared in [Sme89]. There the splitting of the alphabets was more part of the modelling of the systems then of the modelling of the control problem. Here the decision of which events are exogenous is part of the control problem.

The idea of using the reflection operator comes from Tom Verhoeff, see [T.V90] and [T.V91]. The proofs of the properties 5.4, 5.5, and theorem 5.6 are from [T.V90].

6
Distributed control

In the previous chapter the question how to control a single system, given a minimal and maximal wanted exogenous behaviour is answered. Using the reflection operator (which returns the complement of some system) it is easily verified that a solution for the control problem exists and how a solution can be computed.

Although the theory does not assume any structure of the system, i.e., it need not be regular, computations can only be done if the system can be represented using finite state automata. For real life applications these automata may have a very large number of states, which in turn results in large computations. To reduce the number of states we try to control a system by controlling several parts of the system individually.

This approach also leads to better modelling of real life applications as systems in real life are very often distributed, i.e., consist of several parts located at different physical places. As an example in this chapter we consider the Alternating Bit Protocol, used by two systems to communicate messages. Each of the two systems is located at a separate location and a transmission wire is used for communication, i.e., for cooperation of the events of each of the systems.

Distributed control can be divided into a number of different approaches, both in the wanted constraints and in the needed control:

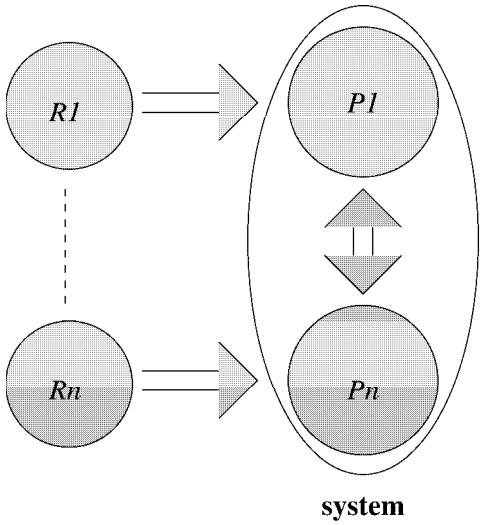
- Control may be *local*: each system is controlled individually; or *global*: one overall controller is given for the system as a whole.
- Wanted constraints may be *local*: desired behaviour is given for each of the subsystems individually; or *global*: one overall desired behaviour for the whole system.

Global control means solving the control problem as is stated and solved in chapter 5. It does not matter here if the constraints are global (the general problem definition) or local. In the second case we first compute global constraints out of the local ones and then solve the general problem.

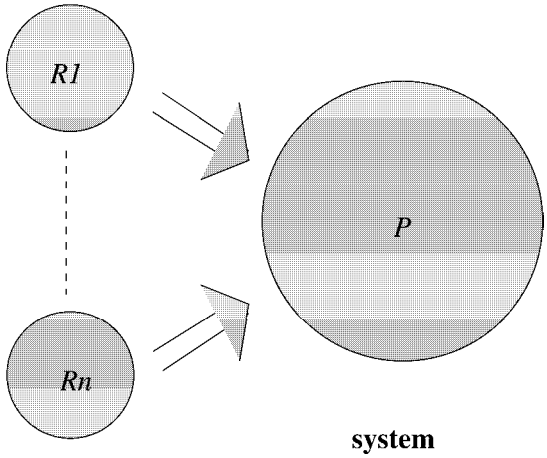
6.1 Distributed Control

In this chapter we will investigate distributed control, i.e., local control with global constraints. A number of possible configurations can be thought of:

- One global system with a number of local controllers



Figuur 6.1: Local systems, local control



Figuur 6.2: Global system, local control

- Local subsystems working in cooperation with each other, where each local subsystem has to be controlled individually.

Having a global system with local control means having one system P and (say n) controllers R_i $(i = 1, \ldots, n)$ such that P controlled by $R_1, \ldots, R_n$ has predefined desired exogenous behaviour. Each R_i controls part of the events of P, namely $\mathbf{a}R_i$. The remaining events of P, i.e., $\mathbf{e}P$, are the exogenous events that should be controlled to behave according to some constraint. This leads to the following problem formulation:

Definition 6.1 [GsLc] *The* Global System Local Control problem *is defined by:*
Given a system P, n alphabets $\mathbf{c}_iP$ $(i = 1, \ldots, n)$, and $\mathbf{e}P$ and global minimal and maximal constraints $L_{\min}$ and $L_{\max}$ with:[1]

$$\mathbf{c}_iP \subseteq \mathbf{a}P \qquad \mathbf{e}P \subseteq \mathbf{a}P$$
$$(\forall i, j : i \neq j : \mathbf{c}_iP \cap \mathbf{c}_jP = \emptyset)$$
$$\mathbf{e}P = \mathbf{a}P \setminus (\bigcup i :: \mathbf{c}_iP)$$
$$L_{\min} \subseteq L_{\max} \subseteq (\mathbf{e}P)^*$$

find controllers R_i $(i = 1, \ldots, n)$, with alphabets $\mathbf{a}R_i = \mathbf{c}_iP$, such that

$$L_{\min} \subseteq P \rceil\lceil (\|\, i :: R_i) \subseteq L_{\max}$$

□

In the second case we have local (say n) subsystems P_i working in cooperation. Each local system is controlled locally by a controller R_i $(i = 1, \ldots, n)$. The local control results in local exogenous behaviour $P_i \rceil\lceil R_i$. The global exogenous behaviour should be according to some predefined minimal and maximal constraints. This leads to the following problem definition:

[1] For some alphabet A we write $P \subseteq A^*$ instead of $P \subseteq \langle A, A^*, A^* \rangle$. Moreover, n will always denote the number of controllers. In quantifiers domains of the dummies will be ignored if they are in the range $1, \ldots, n$.

Definition 6.2 [LsLc] *The* Local System Local Control problem *is defined by:*
Given systems P_i $(i = 1, \ldots, n)$*, alphabets* $\mathbf{a}_{ij}$*,* $\mathbf{c}P_i$*, and* $\mathbf{e}P_i$ $(i, j = 1, \ldots, n)$ *and global minimal and maximal constraints* $L_{\min}$ *and* $L_{\max}$ *with:*

$$\begin{array}{ll} \mathbf{a}_{ij} = \mathbf{a}P_i \cap \mathbf{a}P_j & \textit{cooperating events} \\ \mathbf{c}P_i \subseteq \mathbf{a}P_i & \textit{control events} \\ \mathbf{a}_{ij} \cap \mathbf{c}P_i = \emptyset & \textit{independency condition} \\ \mathbf{e}P_i = \mathbf{a}P_i \setminus \mathbf{c}P_i & \textit{exogenous events} \\ L_{\min} \subseteq L_{\max} \subseteq (\bigcup i :: \mathbf{e}P_i)^* & \end{array}$$

find controllers R_i $(i = 1, \ldots, n)$ *with* $\mathbf{a}R_i = \mathbf{c}P_i$ *such that*

$$L_{\min} \subseteq (\|\, i :: (P_i \rceil\lceil R_i)) \subseteq L_{\max}$$

❑

Exercise 6.1 Show that we also have:

$$\mathbf{c}P_i \cap \mathbf{e}P_i = \emptyset \qquad i \neq j \Rightarrow \mathbf{c}P_i \cap \mathbf{c}P_j = \emptyset \qquad a_{ij} = a_{ji}$$

❑

The general idea is having a system $(\|\, i :: P_i)$ (i.e., built out of n other systems working in cooperation), to find n separate controllers R_i working on P_i, such that the total system acts as desired.

P_i and P_j cooperate through events $\mathbf{a}P_i \cap \mathbf{a}P_j = \mathbf{e}P_i \cap \mathbf{e}P_j$. We call these events *cooperating events* in the distributed system and denote them by $\mathbf{a}_{ij}$ as in the definition.

The systems P_i can be seen as components of a system situated at different locations. The cooperation between these two systems is done via the cooperating events.

In this chapter we will only address the second problem. The LsLc-problem can be divided into two subproblems, dependent on the constraints in the minmax condition. We will have LsLcLg (Local Systems Local Control Local Goals) and LsLcGg (Local Systems Local Control Global Goals).

6.2 LsLcLg: Local control, local goals

Suppose that

$$L_{\min} = (\|\, i :: L_{i,\min}) \;\wedge\; L_{\max} = (\|\, i :: L_{i,\max})$$

where:

$$\mathbf{a}L_{i,\min} = \mathbf{e}P_i \;\wedge\; \mathbf{a}L_{i,\max} = \mathbf{e}P_i$$

Then the control problem can be written as:

$$(\|\, i :: L_{i,\min}) \subseteq (\|\, i :: (P_i \rceil\lceil R_i)) \subseteq (\|\, i :: L_{i,\max})$$

We ask ourselves the question when this problem is solvable by solving the two separate CODE-problems independently (and visa vica), i.e., when is

$$(\forall i :: L_{i,\min} \subseteq P_i \rceil\lceil R_i \subseteq L_{i,\max})$$

$\Leftrightarrow$

$$(\|\, i :: L_{i,\min}) \subseteq (\|\, i :: (P_i \,\|\, R_i)) \subseteq (\|\, i :: L_{i,\max})$$

If equivalence holds, we have two separate control problems and the LsLcLg-problem can easily be solved by solving n CODE-problems, one for each P_i, namely to find controllers R_i such that

$$L_{i,\min} \subseteq P_i \,||\, R_i \subseteq L_{i,\max}$$

6.2.1 Independent systems

Suppose systems P_i are independent, i.e.,

$$(\forall i, j :: \mathbf{a}_{ij} = \emptyset)$$

In that case we can use the following lemma:

Lemma 6.3

$$\begin{array}{ll} & P_1 \subseteq M_1 \ \wedge \ P_2 \subseteq M_2 \ \wedge \ \mathbf{a}P_1 \cap \mathbf{a}P_2 = \emptyset \\ \Leftrightarrow & \\ & P_1 \,||\, P_2 \subseteq M_1 \,||\, M_2 \ \wedge \ \mathbf{a}P_1 \cap \mathbf{a}P_2 = \emptyset \end{array}$$

proof: ($\Rightarrow$):

$$\begin{array}{ll} & P_1 \subseteq M_1 \ \wedge \ P_2 \subseteq M_2 \\ \Rightarrow & \text{[monotonicity of } || \text{]} \\ & P_1 \,||\, P_2 \subseteq M_1 \,||\, P_2 \ \wedge \ P_2 \subseteq M_2 \\ \Rightarrow & \text{[again]} \\ & P_1 \,||\, P_2 \subseteq M_1 \,||\, M_2 \end{array}$$

($\Leftarrow$):

$$\begin{array}{ll} & P_1 \,||\, P_2 \subseteq M_1 \,||\, M_2 \\ \Rightarrow & \text{[monotonicity of } \lceil \text{]} \\ & (P_1 \,||\, P_2)\lceil \mathbf{a}P_1 \subseteq (M_1 \,||\, M_2)\lceil \mathbf{a}P_1 \\ \Leftrightarrow & \text{[step } (*)\text{, see note]} \\ & P_1\lceil \mathbf{a}P_1 \,||\, P_2\lceil \mathbf{a}P_1 \subseteq M_1\lceil \mathbf{a}P_1 \,||\, M_2\lceil \mathbf{a}P_1 \\ \Rightarrow & [\ P_2\lceil \mathbf{a}P_1 = \langle \emptyset, \emptyset, \emptyset \rangle \ \wedge \ \mathbf{a}P_1 = \mathbf{a}M_1 \ \wedge \ M_2\lceil \mathbf{a}P_1 = \langle \emptyset, \emptyset, \emptyset \rangle \ \wedge \ S \,||\, \langle \emptyset, \emptyset, \emptyset \rangle = S\ ] \\ & P_1\lfloor \mathbf{a}P_1 \subseteq M_1\lfloor \mathbf{a}P_1 \\ \Leftrightarrow & [\ \mathbf{a}P_1 = \mathbf{a}M_1\ ] \\ & P_1 \subseteq M_1 \end{array}$$

and similar for $P_2 \subseteq M_2$.
Note: in step $(*)$ we used that, if $\mathbf{a}R \cap \mathbf{a}S \subseteq A$, then $(R \,||\, S)\lceil A = R\lceil A \,||\, S\lceil A$ (See property T.15). This rule can be applied because $\mathbf{a}P_1 \cap \mathbf{a}P_2 = \emptyset$. □

Theorem 6.4 *If*

$$L_{\min} = (\|\, i :: L_{i,\min}) \ \wedge \ L_{\max} = (\|\, i :: L_{i,\max}) \ \wedge \ (\forall i, j : i \neq j : \mathbf{a}P_i \cap \mathbf{a}P_j = \emptyset)$$

then:

$$(\forall i :: L_{i,\min} \subseteq P_i \rceil\lceil F(P_i, L_{i,\max}))$$
$\Leftrightarrow$
the LsLcLg-problem is solvable

proof: Using the previous lemma:

LsLcLg solvable
$\Leftrightarrow$
$(\exists R_1, \ldots, R_n :: (\|\, i :: L_{i,\min}) \subseteq (\|\, i :: (P_i \rceil\lceil R_i)) \subseteq (\|\, i :: L_{i,\max})$
$\Leftrightarrow$ [lemma and $\mathbf{a}(P_i \rceil\lceil R_i) \cap \mathbf{a}(P_j \rceil\lceil R_j) = \emptyset$ if $i \neq j$]
$(\exists R_1, \ldots, R_n :: (\forall i : L_{i,\min} \subseteq (P_i \rceil\lceil R_i) \subseteq L_{i,\max})$
$\Leftrightarrow$
$(\forall i :: L_{i,\min} \subseteq P_i \rceil\lceil F(P_i, L_{i,\max}))$

□

6.2.2 Cooperating systems

If $\mathbf{a}P_i \cap \mathbf{a}P_j \neq \emptyset$ for $i \neq j$ we cannot use lemma 6.3, but only the ($\Rightarrow$) part. Then we have:

Theorem 6.5 *If*

$$L_{\min} = (\|\, i :: L_{i,\min}) \;\wedge\; L_{\max} = (\|\, i :: L_{i,\max})$$

then:

$$(\forall i :: L_{i,\min} \subseteq P_i \rceil\lceil F(P_i, L_{i,\max}))$$
$\Rightarrow$
the LsLcLg-problem is solvable

proof: We prove that $R_i = F(P_i, L_i)$ is a solution:

$L_{\min}$
$=$ [pre-assumption]
$(\|\, i :: L_{i,\min})$
$\subseteq$ [assumption]
$(\|\, i :: (P_i \rceil\lceil F(P_i, L_{i,\max})))$
$\subseteq$ [CODE-problem]
$(\|\, i :: L_{i,\max})$
$=$ [pre-assumption]
$L_{\max}$

□

6.3 LsLcGg: Local control, global goal

Now consider the situation of global goals, i.e., $L_{\min}$ and $L_{\max}$ are given in general. Because we have:

$$L_{\min} \subseteq (\|\, i :: L_{\min} \lceil \mathbf{e}P_i)$$

we can easily derive the following theorem:

Theorem 6.6 *If* $(\exists L_1, \ldots, L_n : (\forall i :: (\mathbf{a}L_i = \mathbf{e}P_i)) : (\|\, i :: L_i) \subseteq L_{\max})$ *then:*

$(\forall i :: L_{\min} \lceil \mathbf{e}P_i \subseteq P_i \rceil [\!] F(P_i, L_i))$

$\Rightarrow$

the LsLcGg-problem is solvable

proof: We prove that $R_i = F(P_i, L_i)$ is a solution:

$$
\begin{array}{ll}
 & L_{\min} \\
\subseteq & \quad [\text{ definition of } \|\text{: } P \subseteq P \lceil A_1 \,\|\, P \lceil A_2 \text{ if } A_1 \cup A_2 = \mathbf{a}P \,] \\
 & (\|\, i :: L_{\min} \lceil \mathbf{e}P_i) \\
\subseteq & \quad [\text{ lemma }] \\
 & (\|\, i : (P_i \rceil [\!] F(P_i, L_i))) \\
\subseteq & \quad [\text{ CODE-problem }] \\
 & (\|\, i :: L_i) \\
\subseteq & \quad [\text{ pre-assumption }] \\
 & L_{\max}
\end{array}
$$

□

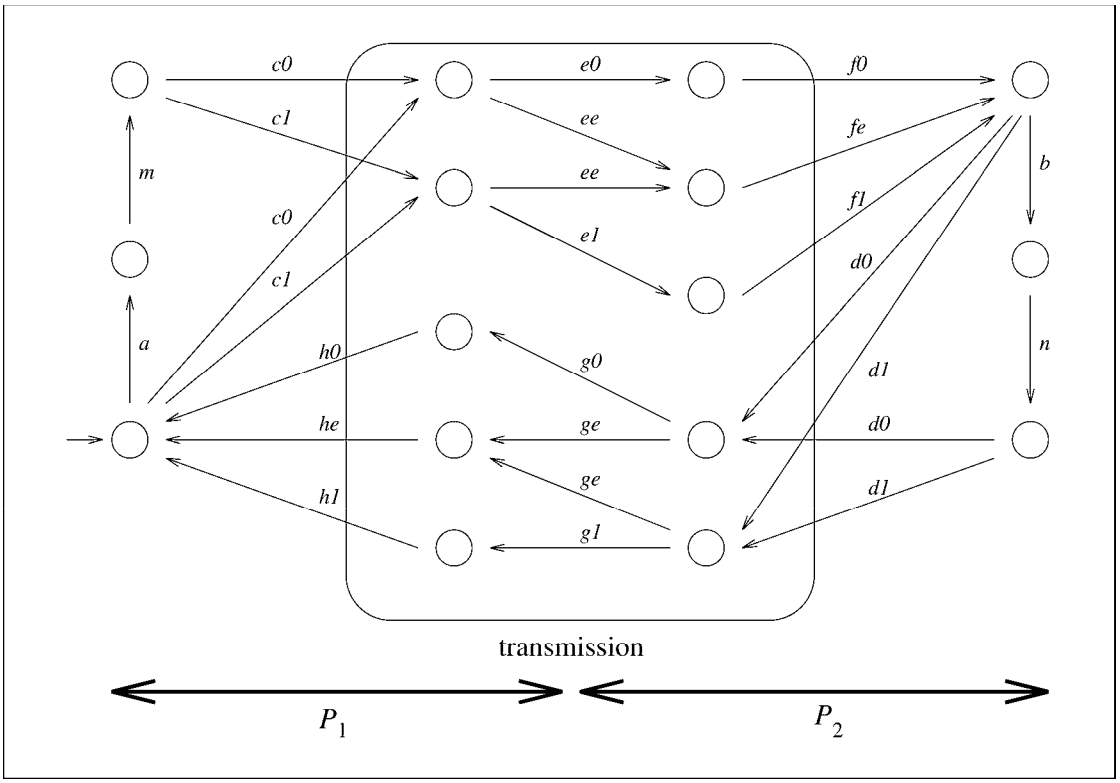
The remaining problem both in this and in the previous section is how to find conditions under which the LsLc-problem can be solved by solving n local CODE problems. The last two theorems only give a solution for the LsLc-problem if the local CODE-problems all solvable for some suitable L_i. In section 6.7 we will return to this remaining problem and try to find a way to compute L_i such that solving the n local CODE problems for these limits results in a solution for LsLc. First however, we will introduce an example to illustrate the LsLc-problem.

6.4 The Alternating Bit Protocol (introduction)

Consider two systems, called *sender* and *receiver*. The sender sends messages to the receiver, one at a time. The sender is allowed to send a next message only after the previous one has been properly received. The receiver acknowledges receiving a message.

Unfortunately, the transmission line between the sender and the receiver is not completely reliable. Sometimes it mutilates messages. We assume, however, that every message sent along the line is received (complete or mutilated) at the other end. We assume some algorithm is present that can tell if a received message is mutilated. In order to be able to send messages along this line, one bit of information is added to the message before it is sent. It turns out that just one bit of extra information is enough to guarantee successful transmission. The configuration is shown in figure 6.3. The meaning of the events is given in table 6.1. To be able to use distributed control we split the system into two systems, P_1 and P_2. The behaviour of P_1 and P_2 is given in figure 6.4 and figure 6.5. It can easily be verified that $P_1 \,\|\, P_2$ is precisely the system presented in figure 6.3. Notice that:

$$
\begin{array}{lcl}
\mathbf{a}_{12} & = & \{e_0, e_1, e_e, g_0, g_1, g_e\} \\
\mathbf{e}P_1 & = & \{e_0, e_1, e_e, g_0, g_1, g_e, m\} \\
\mathbf{e}P_2 & = & \{e_0, e_1, e_e, g_0, g_1, g_e, n\} \\
\mathbf{c}P_1 & = & \{c_0, c_1, a, h_0, h_1, h_e\} \\
\mathbf{c}P_2 & = & \{d_0, d_1, b, f_0, f_1, f_e\}
\end{array}
$$



Figuur 6.3: System configuration

P_1 and P_2 are modelled with as much freedom as possible. It can be checked that $P_1 \,||\, P_2$ equals the system as given in figure 6.3, so indeed P_1 and P_2 are as needed for applying the LsLc-problem.

From the literature (see [BSW69]) we know that this way of data transmission can only be successful if we alternate the value of this extra bit. I.e., we would like to have the exogenous behaviour $L_{\min} = L_{\max}$ as given in figure 6.6. It is beyond the scope of this paper to explain why the exogenous behaviour of figure 6.6 leads to successful transmission and simpler behaviour does not. We only remark that $\mathbf{t}L_{\max}\lceil\{m,n\} = (mn)^*$ (each message is first completely transmitted and handled before a new message is transmitted).

Our problem consists of finding suitable controllers for P_1 and P_2 to establish $L_{\max}$, the desired exogenous behaviour of the whole system. We emphasize that we do not derive the alternating bit protocol here, nor prove its correctness, but only try to find suitable controllers for both sender and receiver, such that the transmission line as a whole behaves according to the alternating bit protocol. This problem is referred to as the ABP-problem.

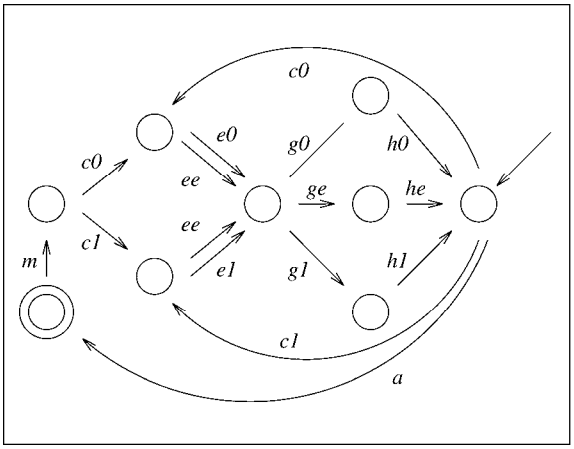
6.5 Solving the ABP-problem (part 0)

First we use CODE to solve the problem as a whole, i.e., we consider $P = P_1 \,||\, P_2$ as one system with $\mathbf{c}P = \mathbf{c}P_1 \cup \mathbf{c}P_2$ and $\mathbf{e}P = \mathbf{e}P_1 \cup \mathbf{e}P_2$. Computing the largest possible

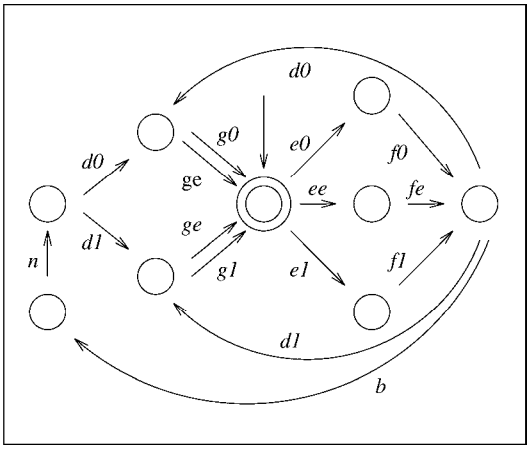
event	meaning	event	meaning
$\mathbf{c}P_1$		$\mathbf{c}P_2$	
c_0	send mesg with 0-flag	d_0	send ack with 0-flag
c_1	send mesg with 1-flag	d_1	send ack with 1-flag
a	ready for next mesg	b	received mesg well
h_0	0-flag ack received	f_0	0-flag mesg received
h_e	ack received with error	f_e	error in receiving mesg
h_1	1-flag ack received	f_1	1-flag mesg received
$\mathbf{e}P_1 \setminus \mathbf{a}_{12}$		$\mathbf{e}P_2 \setminus \mathbf{a}_{12}$	
m	mesg to be sent	n	receiving mesg
$\mathbf{a}_{12}$			
e_0	0-flag mesg transmitted	g_0	0-flag ack transmitted
e_e	mesg transmitted with error	g_e	error in transmitting ack
e_1	1-flag mesg transmitted	g_1	1-flag ack transmitting

mesg = message ack = acknowledgement

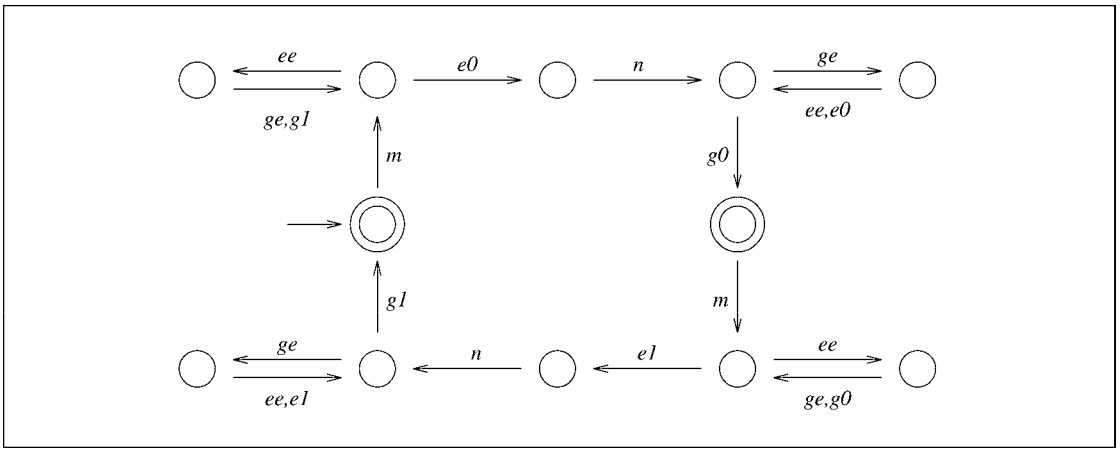
Tabel 6.1: Meaning of the events



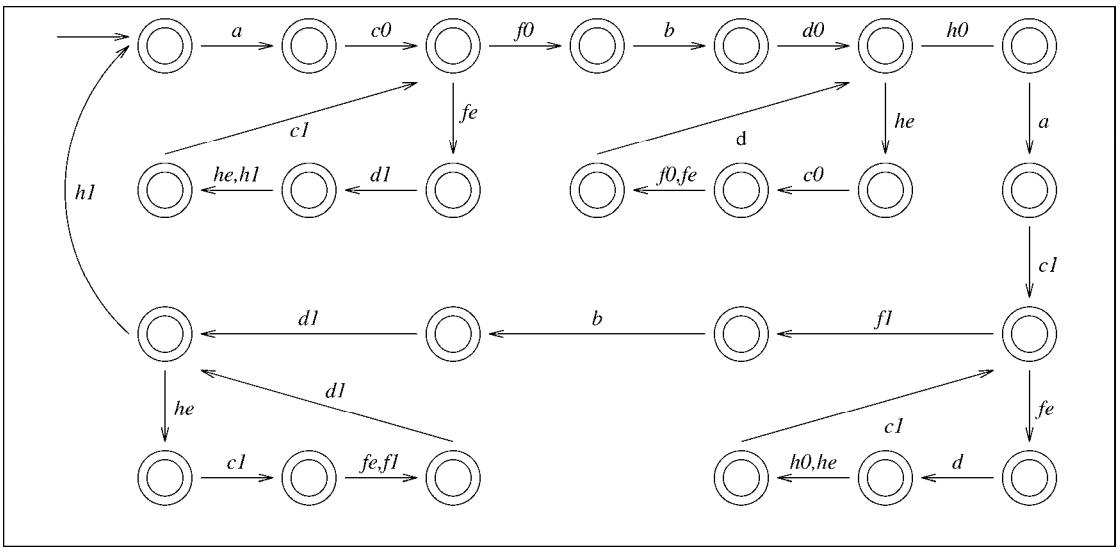
Figuur 6.4: Diagram for system P_1 (sender)



Figuur 6.5: Diagram for system P_2 (receiver)



Figuur 6.6: Desired exogenous behaviour $L_{\max}$



Figuur 6.7: Effective overall-controller $F(P, L_{\max}) \cap P\lceil \mathbf{c}P$.

candidate for a solution:

$$F(P, L_{\max})$$

leads to the system as is given in figure 6.7.[2] It can be verified that indeed

$$L_{\min} \subseteq P \rceil\lceil F(P, L_{\max})$$

so we have a solution for our problem. However, it is not clear at first what this controller does. Furthermore, we like to have separate controllers for P_1 and P_2 such as to get the goal $L_{\max}$. Having separate controllers makes it more likely to have systems that have a smaller size (read: less states) than having one overall-controller, which makes distributed control even more preferable, because for real problems, the automata, representing the systems, may be very large. So we try to solve the ABP-problem using global constraints and local control.

Trying theorem 6.6 on the ABP-problem does not directly lead to success, as is shown in the next section.

6.6 Solving the ABP-problem (part I)

It can be shown that for the ABP-problem we have:

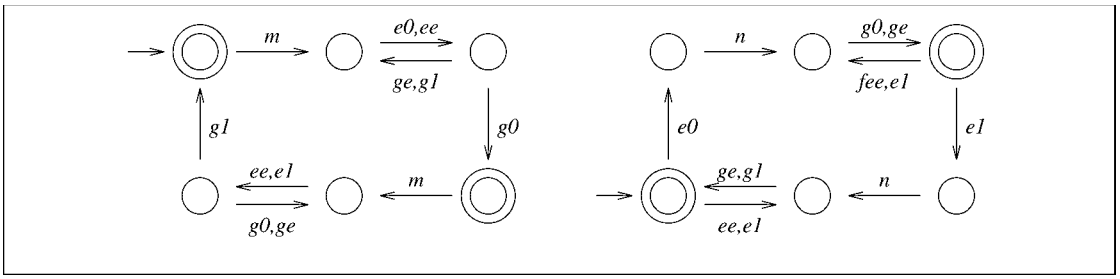
$$L_{\max}\lceil \mathbf{e}P_1 \parallel L_{\max}\lceil \mathbf{e}P_2 = L_{\max}$$

so possible candidates to use for solving the ABP-problem according to theorem 6.6 are

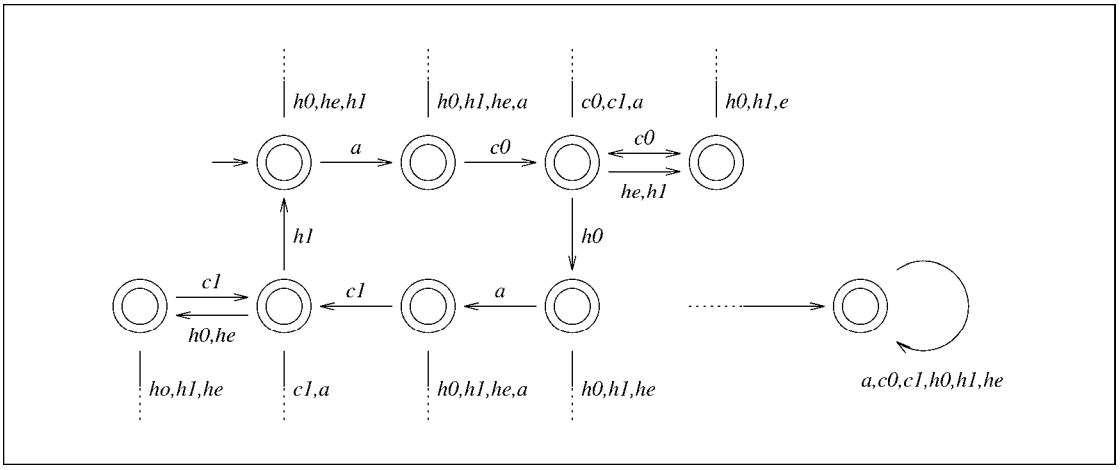
$$L_1 = L_{\max}\lceil \mathbf{e}P_1 \qquad L_2 = L_{\max}\lceil \mathbf{e}P_2$$

However, computing $F(P_1, L_1)$ and $F(P_2, L_2)$ leads to unusable controllers, i.e., the minmax condition with $L_{\min} = L_{\max}$ is not met: $P_i \rceil\lceil F(P_i, L_i) \subset L_i, i = 1, 2$, so the ABP-problem cannot be solved with this L_1 and L_2.

[2] The figure displays the effective part of the controller only.



Figuur 6.8: Useful exogenous behaviour L_1 (left) and L_2 (right) to solve the ABP-problem



Figuur 6.9: Resulting controller R_1 using the behaviours as given in figure 6.8

It is, however, possible to find controllers for P_1 and P_2 such that the behaviour is as described by $L_{\max}$. If we use L_1 and L_2 as given in figure 6.8, we find $L_1 \parallel L_2 = L_{\max}$ and

$$F(P_1, L_1) \rceil\!\lceil P_1 = L_1 \qquad F(P_2, L_2) \rceil\!\lceil P_2 = L_2$$

The controllers are given in figures 6.9 and 6.10. They contain also transitions that do not have any influence on the system. If we use the so-called *effective controllers*:

$$F(P_i, L_i) \cap P_i\lceil \mathbf{c}P_i$$

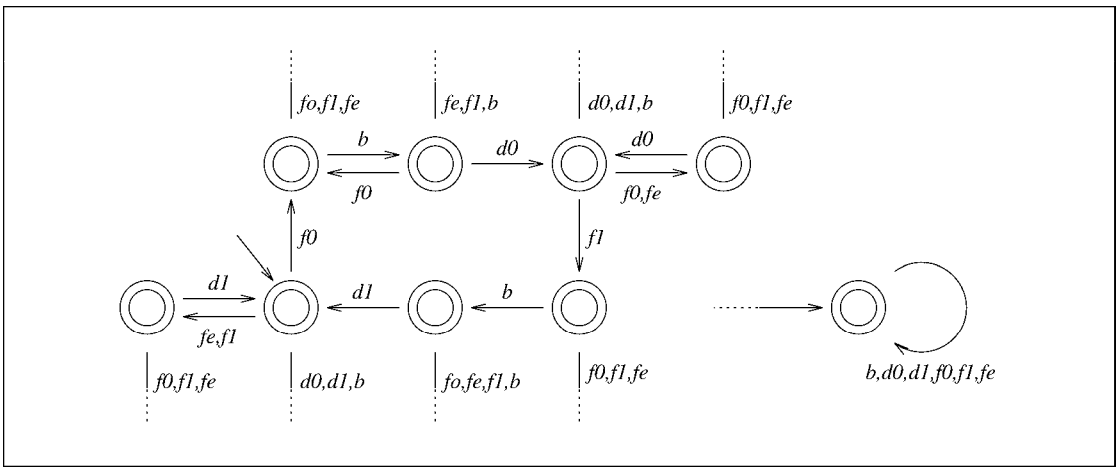
only events that can be applied to the system are considered. The effective versions of the controllers R_1 and R_2 can be found in figure 6.11.

From the ABP-problem and the above solution, we conclude that not all L_1 and L_2 with $L_1 \parallel L_2 = L_{\max}$ give solutions. Notice that $L_{\max}\lceil \mathbf{e}P_1 \subseteq L_1$ and $L_{\max}\lceil \mathbf{e}P_2 \subseteq L_2$, so probably L_1 and L_2 should be as large as possible.

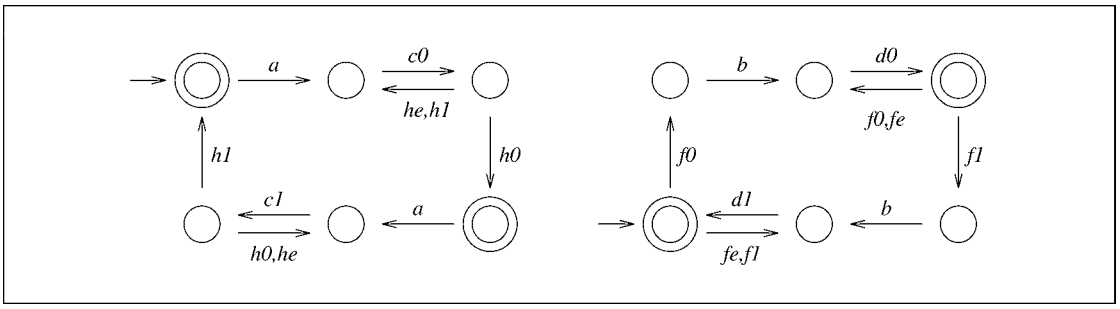
6.7 Non-trivial solution

Assuming that $P_i \rceil\!\lceil R_i \subseteq L_i$ $(i = 1, \ldots, n)$, the following derivation can be made:

the LsLc-problem is solvable



Figuur 6.10: Resulting controller R_2 using the behaviours as given in figure 6.8



Figuur 6.11: Minimized effective controllers R_1 (left) and R_2 (right) computed from the controllers of figures 6.9 and 6.10

$\Leftrightarrow$
$\quad (\exists R_1, \ldots, R_n :: L_{\min} \subseteq (\|\, i :: (P_i \,]\!|\, R_i)) \subseteq L_{\max})$
$\Rightarrow$
$\quad (\exists R_1, \ldots, R_n :: L_{\min}\lceil \mathbf{e}P_i \subseteq (\|\, i : ((P_i \,]\!\lceil\, R_i)))\lceil \mathbf{e}P_i)$
$\Rightarrow$ [$(S_1 \parallel S_2)\lceil \mathbf{a}S_1 \subseteq S_1$]
$\quad (\exists R_1, \ldots, R_n :: L_{\min}\lceil \mathbf{e}P_i \subseteq P_i \,]\!\lceil\, R_i)$
$\Rightarrow$ [by assumption]
$\quad (\exists R_i :: L_{\min}\lceil \mathbf{e}P_i \subseteq P_i \,]\!|\, R_i \subseteq L_i)$
$\Leftrightarrow$ [CODE-problem]
$\quad L_{\min}\lceil \mathbf{e}P_i \subseteq P_i \,]\!\lceil\, F(P_i, L_i)$

Using this result and theorem 6.6 we may conclude that the LsLc-problem can be solved by solving CODE n times only if we can find L_i such that each solution R_i has the property that $P_i \,]\!\lceil\, R_i \subseteq L_i$, or (by reformulating)

$$(\forall S_1, \ldots, S_n : L_{\min} \subseteq (\|\, i :: S_i) \subseteq (\|\, i :: L_i) \subseteq L_{\max} : (\forall i :: S_i \subseteq L_i))$$

At this point we can conclude that the LsLc-problem can be solved by solving n CODE problems only if the above condition is true. So we should find L_i with $(\|\, i :: L_i) \subseteq L_{\max}$

such that each solution R_i has the property $P_i \,||\, R_i \subseteq L_i$, i.e., we should find the largest such L_i.

A necessary and sufficient condition for the LsLc-problem to have a solution is given in the next lemma.

Lemma 6.7 *For L_i^m with $(\|\, i :: L_i^m) \subseteq L_{\max}$ and*

$$(\forall L_1, \ldots, L_n : (\|\, i :: L_i) \subseteq L_{\max} : (\forall i :: L_i \subseteq L_i^m))$$

we have that:

the LsLc-problem is solvable

$\Leftrightarrow$

$$(\forall i :: L_{\min}\lceil \mathbf{e}P_i \subseteq P_i \rceil\lceil\, F(P_i, L_i^m))$$

proof: That the condition is sufficient can be found in theorem 6.6. That it is necessary follows from the previous derivation. □

This lemma is useful only if L_i^m can be found. Therefore, we first consider the problem of finding such systems.

6.8 Separation of systems

In this section we try to solve the following problem:

Definition 6.8 [Separation problem]
Given is a DES L and n languages $A_1, \ldots, A_n$. To find the n largest systems L_i such that

$$(\forall i :: L_i \subseteq A_i^*) \;\wedge\; (\|\, i :: L_i) = L$$

□

According to [WM91] we call a system L *separable* with respect to alphabets $(A_1, \ldots, A_n)$ if

$$(\exists L_1, \ldots, L_n : (\forall i :: L_i \subseteq A_i^*) : (\|\, i :: L_i) = L)$$

It should be clear that L is only separable w.r.t. $(A_1, \ldots, A_n)$ if $(\bigcup i :: A_i) = \mathbf{a}L$.

Exercise 6.2 Is $L = \langle \{a, b, c\}, ab|ba \rangle$ separable w.r.t. $(\{a, c\}, \{b, c\})$? If so, give a separation for L.
Answer the same questions for $L' = \langle \{a, b, c\}, ab \rangle$. □

Definition 6.9 *A set $(L_1, \ldots, L_n)$ with $(\|\, i :: L_i) = L$ is called a* generating set *for L. We will collect all generating sets (w.r.t. some predefined alphabets) in $B(L)$, e.g.,*

$$B(L) = \{L_1, \ldots, L_n : (\forall i :: \mathbf{a}L_i = A_i) \;\wedge\; (\|\, i :: L_i) = L : (L_1, \ldots, L_n)\}$$

With $B_i(L)$ we denote the family of i-th components of generating sets of L, e.g.,

$$B_i(L) = \{L_1, \ldots, L_n : (\forall i :: \mathbf{a}L_i = A_i) \;\wedge\; (\|\, i :: L_i) = L : L_i\}$$

□

The following properties should be clear:

Property 6.10

$(a)\ (\|\!|\, i : (\bigcup S : S \in B_i(L) : S)) \supseteq L$
$(b)\ (\|\!|\, i : (\bigcap S : S \in B_i(L) : S)) \subseteq L$

□

Property 6.11 *The set $B(L)$ is closed under arbitrary intersection.*
proof: [WM91] Let (for each i) $\{S_i^1, \ldots, S_i^q\}$ be some subset of elements of $B_i(L)$, such that $(\forall j : 1 \leq j \leq q : (\|\!|\, i :: S_i^j) = L)$. We need to show that the intersection of these subsets is also in $B_i(L)$ for each i:

$$
\begin{array}{ll}
 & (\|\!|\, i :: (\bigcap j : 1 \leq j \leq q : S_i^j)) \\
= & \quad [\ \mathbf{a}S_i^j = \mathbf{a}S_i^k\ ] \\
 & (\|\!|\, i :: (\|\!|\, j : 1 \leq j \leq q : S_i^j)) \\
= & \\
 & (\|\!|\, j : 1 \leq j \leq q : (\|\!|\, i :: S_i^j)) \\
= & \quad [\ (S_1^j, \ldots, S_n^j) \text{ is a generating set }\ ] \\
 & (\|\!|\, j : 1 \leq j \leq q : L) \\
= & \\
 & L
\end{array}
$$

So $((\bigcap j : 1 \leq j \leq q : S_1^j), \ldots, (\bigcap j : 1 \leq j \leq q : S_n^j))$ is also a generating set. □

Example 6.12 Let $L = \langle \{a, b, c, d\}, cab|cba \} \rangle$, $A_1 = \{a, c, d\}$, and $A_2 = \{b, c, d\}$. Then L is separable w.r.t. (A_1, A_2). Let

$$
\begin{array}{ll}
L_1^1 = \langle \{a, b, d\}, ca|da \rangle & L_2^1 = \langle \{b, c, d\}, cb|cd \rangle \\
L_1^2 = \langle \{a, b, d\}, ca \rangle & L_2^2 = \langle \{b, c, d\}, cb|dc|db \rangle
\end{array}
$$

Then (L_1^1, L_2^1) and (L_1^2, L_2^2) are both generating sets, but, according to the previous property, also $(L_1^1 \cap L_1^2, L_2^1 \cap L_2^2)$ is a generating set. Notice that:

$$L_1^1 \cap L_1^2 = \langle \{a, b, d\}, ca \rangle \qquad L_2^1 \cap L_2^2 = \langle \{b, c, d\}, cb \rangle$$

□

Property 6.13

$$(\forall i :: (\bigcap S : S \in B_i(L) : S) = L\lceil A_i)$$

proof: [WM91]

$$
\begin{array}{ll}
 & L\lceil A_i \\
= & \quad [\ \text{property 6.11: } (\bigcap S : S \in B_i(L) : S) \in B_i(L)\ ] \\
 & (\|\!|\, j :: (\bigcap S : S \in B_j(L) : S))\lceil A_i \\
\subseteq & \quad [\ (P \,\|\, R)\lceil \mathbf{a}P \subseteq P\ ] \\
 & (\bigcap S : S \in B_i(L) : S)
\end{array}
$$

So we have $L\lceil A_i \subseteq (\bigcap S : S \in B_i(L) : S)$. Moreover,

$$
\begin{array}{ll}
 & L \\
\subseteq & \quad [\text{ for } A_1 \cup A_2 = \mathbf{a}P \text{ we have } P \subseteq P\lceil A_1 \parallel P\lceil A_2 \ ] \\
 & (\|\, i :: L\lceil A_i) \\
\subseteq & \quad [\text{ monotonicity of } \parallel \text{ and } L\lceil A_i \subseteq (\bigcap S : S \in B_i(L) : S) \ ] \\
 & (\|\, i :: (\bigcap S : S \in B_i(L) : S)) \\
= & \quad [\text{ property 6.11 }] \\
 & L
\end{array}
$$

So we have

$$
\begin{array}{l}
L = (\|\, i :: L\lceil A_i) \\
\Rightarrow \\
(\forall i :: L\lceil A_i \in B_i(L)) \\
\Rightarrow \\
(\forall i :: L\lceil A_i \supseteq (\bigcap S : S \in B_i(L) : S))
\end{array}
$$

So we also have $L\lceil A_i \supseteq (\bigcap S : S \in B_i(L) : S)$. □

Example 6.14 Reconsider example 6.12. We have $L\lceil A_1 = \langle \{a, c, d\}, ca \rangle$ and $L\lceil A_2 = \langle \{b, c, d\}, cb \rangle$ and indeed $L_1^1 \cap L_1^2 = L\lceil A_1$ and $L_2^1 \cap L_2^2 = L\lceil A_2$ and $L\lceil A_1 \parallel L\lceil A_2 = L$. □

Property 6.15

(a) L is separable $\Leftrightarrow L = (\|\, i :: L\lceil A_i)$
(b) $(\forall j :: L = (\|\, i :: L_i) \Rightarrow L = (\|\, i : i \neq j : L_i) \parallel L\lceil A_j)$

proof: [WM91] (a) is obvious. For (b):

$$
\begin{array}{ll}
 & L \\
= & \quad [\text{ part (a) and } L \text{ is separable }] \\
 & (\|\, i :: L\lceil A_i) \\
= & \\
 & (\|\, i : i \neq j : L\lceil A_i) \parallel L\lceil A_j \\
\subseteq & \quad [\text{ Property 6.13: } L\lceil A_j = (\bigcap S : S \in B_i(L) : S) \subseteq L_i \ ] \\
 & (\|\, i : i \neq j : L_i) \parallel L\lceil A_j \\
\subseteq & \quad [\text{ again }] \\
 & (\|\, i : i \neq j : L_i) \parallel L_j \\
= & \\
 & L
\end{array}
$$

□

(a) states that we can test if a system is separable by just computing $(\|\, i :: L\lceil A_i)$ and see if this equals L itself. (b) implies that we can replace an arbitrary number of systems L_i from the generating set by $L\lceil A_i$, i.e.,

$$
\begin{array}{ll}
 & L \\
= & (\|\, i :: L_i) \\
= & (\|\, i : i \in J : L_i) \parallel (\|\, i : i \notin J : L\lceil A_i)
\end{array}
$$

Especially we have:

Property 6.16 *If* $S \in B_i(L)$ *then* $L = (\|\, j : j \neq i : L\lceil A_j) \,\|\, S$ □

Example 6.17 Reconsider example 6.12. Also $(L_1^1, L\lceil A_2)$, $(L_1^2, L\lceil A_2)$, $(L\lceil A_1, L_2^1)$, and $(L\lceil A_1, L_2^2)$ are generating sets for L. □

Example 6.18 Reconsider example 6.12. Simply do a element-wize union of all generating sets does not result in a generating set:

$$L_1^1 \cup L_1^2 = \langle \{a, b, d\}, ca|da \rangle \qquad L_2^1 \cup L_2^2 = \langle \{b, c, d\}, cb|cd|dc|db \rangle$$

and $(L_1^1 \cup L_1^2, L_2^1 \cup L_2^2)$ is no longer a generating set. For example dac, dca, dab and more tasks are not in L. □

Property 6.19 *For each* q *and* i*:*

$$\begin{array}{l} (\forall j : 1 \leq j \leq q \ \wedge \ S_i^j \in B_i(L) : (\|\, k : k \neq i : L_k) \,\|\, S_i^j = L) \\ \Rightarrow \\ (\|\, k : k \neq i : L_k) \,\|\, (\bigcup j : 1 \leq j \leq q \ \wedge \ S_i^j \in B_i(L) : S_i^j) = L) \end{array}$$

proof: [WM91]

$$\begin{array}{ll} & (\|\, k : k \neq i : L_k) \,\|\, (\bigcup j : 1 \leq j \leq q \ \wedge \ S_i^j \in B_i(L) : S_i^j)) \\ = & \quad [\text{ property 1.23 (a) }] \\ & (\bigcup j : 1 \leq j \leq q \ \wedge \ S_i^j \in B_i(L) : (\|\, k : k \neq i : L_k) \,\|\, S_i^j) \\ = & \quad [\text{ given }] \\ & (\bigcup j : 1 \leq j \leq q \ \wedge \ S_i^j \in B_i(L) : L) \\ = & \\ & L \end{array}$$

□

This property states that if we have chosen all but one of the separate elements of some generating set we can find the greatest remainder element by taking the union of all elements that, together with the already chosen elements, form a generating set.

Example 6.20 Reconsider example 6.12. Not only (L_1^2, L_2^1) and (L_1^2, L_2^2) are a generating sets, but, according to the previous property, also $(L_1^1, L_2^1 \cup L_2^2)$. □

The following property gives a way of computing such a greatest element without having to know all possible elements of which $B_i(L)$ that should be united.

Definition 6.21 *Define for each generating set* $(L_1, \ldots, L_n)$ *the operator* Q_i *as follows:*

$$Q_i(L_1, \ldots, L_n) = L_i \cup {\sim}(\tilde{L}_i\lceil A_i)$$

with

$$\tilde{L}_i = (\|\, j : j \neq i : L_j) \,\|\, A_i^*$$

□

Now we have the following property, stating that $Q_i(L_1, \ldots, L_n)$ computes the largest possible replacement for L_i in the generating, such that the set

$$(L_1, \ldots, \underbrace{Q_i(L_1, \ldots, L_n)}_{i\text{-th position}}, \ldots, L_n)$$

is still a generating set.

Property 6.22 *Let $(L_1, \ldots, L_n)$ be a given generating set. Let $B(L_i)$ be the set of all systems S such that $(L_1, \ldots, S, \ldots, L_n)$ is also a generating set (i.e., L_i replaced by S). Then we have:*

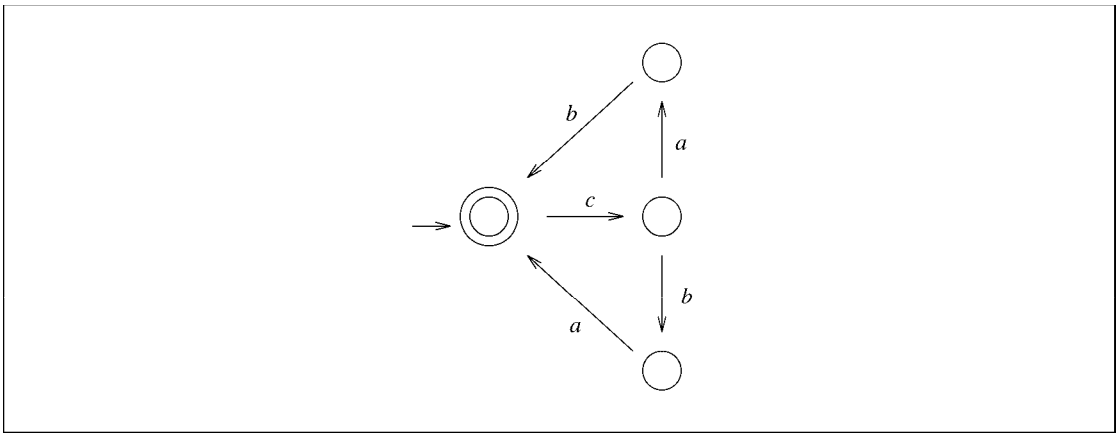
$$(\bigcup S : S \in B(L_i) : S) = Q_i(L_1, \ldots, L_n)$$

proof: [WM91]
Use Q_i as shorthand for $Q_i(L_1, \ldots, L_n)$.
First we prove that $Q_i \subseteq (\bigcup S : S \in B(L_i) : S)$ as follows: Let $S_i = (\|\, j : j \neq i : L_j)$, then:

$$\begin{array}{ll}
& S_i \parallel Q_i \\
= & \quad [\text{ definition of } Q_i \text{ and property 1.26 (b) }] \\
& (\|\, j :: L_j) \cup (S_i \parallel {\sim}(\tilde{L}_i \lceil A_i)) \\
= & \quad [\text{ definition of } \sim \,] \\
& L \cup (S_i \parallel (A_i^* \backslash \tilde{L}_i \lceil A_i)) \\
= & \quad [\text{ property 1.23 (c) }] \\
& L \cup ((S_i \parallel A_i^*) \backslash (S_i \parallel \tilde{L}_i \lceil A_i)) \\
= & \quad [\text{ definition of } \tilde{L}_i \,] \\
& L \cup ((S_i \parallel A_i^*) \backslash (S_i \parallel (S_i \parallel A_i^*) \lceil A_i)) \\
= & \quad [\text{ property 1.21 (b): } \mathbf{a}S_i \cap A_i \subseteq A_i \,] \\
& L \cup ((S_i \parallel A_i^*) \backslash (S_i \parallel S_i \lceil A_i \parallel A_i^*)) \\
= & \quad [\, P \parallel P \lceil A = P \,] \\
& L \cup ((S_i \parallel A_i^*) \backslash (S_i \parallel A_i^*)) \\
= & \\
& L
\end{array}$$

So $Q_i \in B(L_i)$ and hence $Q_i \subseteq (\bigcup S : S \in B(L_i) : S)$.
Next we prove $Q_i \supseteq (\bigcup S : S \in B(L_i) : S)$ as follows:

$$\begin{array}{ll}
& t \in \mathbf{b}(\bigcup S : S \in B(L_i) : S) \;\wedge\; t \notin \mathbf{b}Q_i \\
= & \quad [\text{ definition of } Q_i \,] \\
& (\exists S : S \in B(L_i) : t \in \mathbf{b}S) \;\wedge\; t \notin \mathbf{b}L_i \;\wedge\; t \in \mathbf{b}\tilde{L}_i \lceil A_i \\
= & \\
& (\exists S : S \in B(L_i) : t \in \mathbf{b}S) \;\wedge\; t \notin \mathbf{b}L_i \;\wedge\; (\exists u : u \in \mathbf{b}\tilde{L}_i : u \lceil A_i = t) \\
\Rightarrow & \\
& (\exists S : S \in B(L_i) : (\exists u : u \in \mathbf{b}\tilde{L}_i : u \lceil A_i \in \mathbf{b}S \;\wedge\; u \lceil A_i \notin \mathbf{b}L \lceil A_i)) \\
= & \\
& (\exists S : S \in B(L_i) : (\exists u : u \in \mathbf{b}(\tilde{L}_i \parallel S) : u \lceil A_i \notin \mathbf{b}L \lceil A_i)) \\
= & \quad [\, \tilde{L}_i \parallel S = L \text{ if } S \in B(L_i) \,] \\
& (\exists S : S \in B(L_i) : (\exists u : u \in \mathbf{b}L_i : u \lceil A_i \notin \mathbf{b}L \lceil A_i))
\end{array}$$



Figuur 6.12: System L for exercise 6.3.

=

false

The same derivation can be made with **b** replaced by **t**, which proves that $Q_i \supseteq (\bigcup S : S \in B(L_i) : S)$. □

If we keep all but one elements of a generating set unchanged, we can compute now the largest element such that we still have a generating set.

Exercise 6.3 Consider again $A_1 = \{a, c, d\}$ and $A_2 = \{b, c, d\}$ and take L as in figure 6.12. First compute the smallest possible generating set. Next compute $L_{11} = Q_1(L_1, L_2)$ and $L_{22} = Q_2(L_{11}, L_2)$. What can be said about L_{22}. Also, compute $L'_{22} = Q_2(L_1, L_2)$ and $L'_{11} = Q_1(L_1, L'_{22})$. What can now be said about L'_{11}? □

6.9 Solving the ABP-problem (part II)

We can use the theory of separation of systems on the ABP-problem now. First we compute the smallest generating set for $L_{\max}$, i.e.,

$$(L_{\max}\lceil \mathbf{e}P_1, L_{\max}\lceil \mathbf{e}P_2)$$

In section 6.6 we have concluded already that $F(P_1, L_{\max}\lceil \mathbf{e}P_1)$ and $F(P_2, L_{\max}\lceil \mathbf{e}P_2)$ do not lead to useful controllers. A next try is to compute the largest possible generating sets, i.e., compute

$$L_1 = Q_1(L_{\max}\lceil \mathbf{e}P_1, L_{\max}\lceil \mathbf{e}P_2)$$
$$L_2 = Q_2(L_{\max}\lceil \mathbf{e}P_1, L_{\max}\lceil \mathbf{e}P_2)$$

As we have seen in exercise 6.3 we cannot use L_1 and L_2 at the same time, because (L_1, L_2) is no longer a generating set. Also largest computing the partner of L_1 to form a generating set leads to $L_{\max}\lceil \mathbf{e}P_2$ again and computing the largest partner of L_2 leads to $L_{\max}\lceil \mathbf{e}P_1$ and these partners do not lead to useful controllers, as is concluded earlier.

Therefore, using this principle does not work.

6.10 Concluding results

The theory of separation only gives results for the LsLc-problem, if:

- If the each of the local CODE problems (for P_i, $L_{i,\min}$) can be solved using $L_{\max}\lceil \mathbf{e}P_i$, we also have solvability of the LsLc-problem.
- If none of the local CODE problems can be solved using $Q_i(L_{\max}\lceil \mathbf{e}P_1, \ldots, L_{\max}\lceil \mathbf{e}P_n)$ as upper limit, no solution for LsLc can be found

Our last theorem, therefore, is:

Theorem 6.23 *Associated with the LsLc-problem we have:*

$$(\forall i :: L_{min}\lceil \mathbf{e}P_i \subseteq P_i \rceil\!\lceil F(P_i, L_{max}\lceil \mathbf{e}P_i))$$
$$\Rightarrow$$
the LsLc-problem is solvable

and

$$(\exists i :: L_{min}\lceil \mathbf{e}P_i \not\subseteq P_1 \rceil\!\lceil F(P_i, Q_i))$$
$$\Rightarrow$$
the LsLc-problem is not solvable

where Q_i is shorthand for $Q_i(L_{\max}\lceil \mathbf{e}P_1, \ldots, L_{\max}\lceil \mathbf{e}P_n)$. □

If neither one of the conditions in this theorem is met, we cannot conclude solvability or unsolvability of the LsLc-problem. In that case, we have to do some tedious hand-work and try to find suitable local upper limits L_i that leads to solvability of the local CODE-problems.

6.11 Conclusions

In this chapter we have defined different approaches to distributed control. We have tried to solve the LsLc-problem and only partly succeeded in that. We found necessary conditions for the LsLc-problem to have a solution and sufficient conditions. Thus we are able to sometimes tell that the problem has no solution at all and sometimes are able to find a solution immediately. In general, however, we have to do tedious hand-work to find controllers.

Other approaches

Discrete event systems are also studied by Ramadge and Wonham [RW89]. Their supervisory control, however, uses an observer system as a controller. This so-called supervisor observes the system and enables or disables controllable events in order to get desired behaviour. Rudie and Wonham give another approach for distributed control (see [RW92]): they use observable events to decide to enable or disable controllable events in order to have a predefined observable behaviour. In our terminology this should mean that the controller could observe the exogenous events only and enables or disables the controllable events in some way such as to get a predefined desired exogenous behaviour. Rudie and Wonham give a condition on the desired observable behaviour (called

co-observability). If this condition is met for some constraint between the given limits, the problem is solvable.

Our approach defines exogenous events to be unobservable. Controllable events are used to observe as well as to control. This leads to another approach called synchronous control. Events in the system are enabled if the corresponding controller can engage in that event.

6.12 References

The theory of separation of systems is adopted from [WM91]. A similar problem derivation appeared in [Sme89], but only for two systems.

This chapter describes just a first attempt to do distributed control. Thanks go to Jan Jongejan and Jos Roerdink for reading an earlier version of this chapter and pointing out some errors and improvements.

Appendix T
Trace theory

In this appendix we give the basic notions on which the material in the report is based, i.e., we discuss some of the aspects of trace theory. For a more detailed description of trace theory we refer to [Sne85].

T.1 Trace structures

The pair consisting of the alphabet A and the trace set S is called a *trace structure* and denoted by[1]

$$T = \langle A, S \rangle$$

We introduce two operators in order to obtain the alphabet and the trace set of some given trace structure:

Definition T.1 *For trace structure* $T = \langle S, A \rangle$ *the operators* **t** *and* **a** *are defined by:*

$$\mathbf{a}T = A \qquad \mathbf{t}T = S$$

□

T.2 Alphabet restriction

Definition T.2 *Alphabet restriction* $\lceil$ *is defined on a trace by:*

$$\begin{array}{lcll} \epsilon\lceil A & = & \epsilon & \\ (ta)\lceil A & = & t\lceil A & \text{if } a \notin A \\ & = & (t\lceil A)a & \text{if } a \in A \end{array}$$

and on trace structures by:

$$T\lceil A = \langle \mathbf{a}T \cap A, \{t : t \in \mathbf{t}T : t\lceil A\} \rangle$$

□

T.3 Weaving of trace structures

Definition T.3 *The weaving* $\underline{\mathbf{w}}$ *of two trace structures* T *and* S *is defined by*

[1] In [Sne85] S and A are interchanged.

$$
\begin{array}{l}
\quad T \underline{\mathbf{w}}\, S \\
= \\
\quad \langle \mathbf{a}T \cup \mathbf{a}S, \{x : x \in (\mathbf{a}T \cup \mathbf{a}S)^* \ \wedge \ x\lceil \mathbf{a}T \in \mathbf{t}T \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\} \rangle
\end{array}
$$

□

The operator $\underline{\mathbf{w}}$ defines a binary operation on the set of trace structures. It has some nice properties, which are listed below:

Property T.4 *For general trace structures T, U, and S, the following hold:*

(1) $T \underline{\mathbf{w}}\, S = S \underline{\mathbf{w}}\, T$
(2) $T \underline{\mathbf{w}}\, \langle \{\epsilon\}, \emptyset \rangle = T$
(3) $T \underline{\mathbf{w}}\, \langle \emptyset, \emptyset \rangle = \langle \emptyset, \emptyset \rangle$
(4) $T \underline{\mathbf{w}}\, T = T$
(5) $(T \underline{\mathbf{w}}\, S) \underline{\mathbf{w}}\, U = T \underline{\mathbf{w}}\, (S \underline{\mathbf{w}}\, U)$

□

Definition T.5 *The blending $\underline{\mathbf{b}}$ of two trace structures T and S is defined by*

$$
\begin{array}{l}
\quad T \underline{\mathbf{b}}\, S \\
= \\
\quad \langle \mathbf{a}T \div \mathbf{a}S, \{x : x \in (\mathbf{a}T \cup \mathbf{a}S)^* \ \wedge \ x\lceil \mathbf{a}T \in \mathbf{t}T \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\lceil (\mathbf{a}T \div \mathbf{a}S)\} \rangle
\end{array}
$$

□

The operator $\div$ stands for symmetric set difference, i.e.,

$$A \div B = (A \cup B) \backslash (A \cap B)$$

Also the operator $\underline{\mathbf{b}}$ is a binary operator on the set of trace structures. Some properties of the blend are listed below:

Property T.6 *For general trace structures T, U, and S, the following hold:*

(1) $T \underline{\mathbf{b}}\, S = S \underline{\mathbf{b}}\, T$
(2) $T \underline{\mathbf{b}}\, \langle \{\epsilon\}, \emptyset \rangle = T$
(3) $T \underline{\mathbf{b}}\, \langle \emptyset, \emptyset \rangle = \langle \emptyset, \emptyset \rangle$
(4) $\mathbf{t}T \neq \emptyset \ \Rightarrow \ T \underline{\mathbf{b}}\, T = \langle \{\epsilon\}, \emptyset \rangle$

□

Blending is not associative. However, we have:

Property T.7 *Blending is associative if every symbol occurs in at most two of the alphabets.* □

Property T.8 *For trace structures T and S with $\mathbf{a}S \subseteq \mathbf{a}T$, the following properties hold:*

$$
\begin{array}{lll}
(1) & T \underline{\mathbf{w}}\, S & = \langle \mathbf{a}T, \{x : x \in \mathbf{t}T \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\} \rangle \\
(2) & T \underline{\mathbf{b}}\, S & = \langle \mathbf{a}T \backslash \mathbf{a}S, \{x : x \in \mathbf{t}T \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\lceil(\mathbf{a}T \backslash \mathbf{a}S)\} \rangle \\
(3) & T \underline{\mathbf{b}}\, S & = \langle \mathbf{a}T \backslash \mathbf{a}S \\
 & & \quad , \{x : x \in \mathbf{t}T\lceil(\mathbf{a}T \backslash \mathbf{a}S) \wedge \\
 & & \qquad (\exists y : y \in \mathbf{t}T \ \wedge \ y\lceil \mathbf{a}S \in \mathbf{t}S : y\lceil(\mathbf{a}T \backslash \mathbf{a}S) = x) \\
 & & \qquad : x\} \\
 & & \quad \rangle \\
 & & = \langle \mathbf{a}T \backslash \mathbf{a}S \\
 & & \quad , \{x : x \in \mathbf{t}T\lceil(\mathbf{a}T \backslash \mathbf{a}S) \wedge \\
 & & \qquad (\exists y : y \in \mathbf{t}T \ \wedge \ y\lceil(\mathbf{a}T \backslash \mathbf{a}S) = x : y\lceil \mathbf{a}S \in \mathbf{t}S) \\
 & & \qquad : x\} \\
 & & \quad \rangle
\end{array}
$$

□

T.4 Ordering of trace structures

Definition T.9 *For two trace structures T and S, the ordering $T \subseteq S$ is defined by:*

$$\mathbf{a}T = \mathbf{a}S \ \wedge \ \mathbf{t}T \subseteq \mathbf{t}S$$

□

Property T.10 *For trace structures T, S_1, and S_2 with $\mathbf{a}S_1 = \mathbf{a}S_2$ (for $i = 1, 2$), we have:*

$$
\begin{array}{ll}
(1) & S_1 \subseteq S_2 \Rightarrow (T \underline{\mathbf{w}}\, S_1) \subseteq (T \underline{\mathbf{w}}\, S_2) \\
(2) & S_1 \subseteq S_2 \Rightarrow (T \underline{\mathbf{b}}\, S_1) \subseteq (T \underline{\mathbf{b}}\, S_2)
\end{array}
$$

□

Lemma T.11 *For trace structures T and S with $\mathbf{a}S \subseteq \mathbf{a}T$ and $S \subseteq T\lceil \mathbf{a}S$, the following holds:*

$$T \underline{\mathbf{b}}\, (T \underline{\mathbf{b}}\, S) \supseteq S$$

proof: Use $A = \mathbf{a}T \backslash \mathbf{a}S$, then

$$
\begin{array}{ll}
 & \mathbf{t}(T \underline{\mathbf{b}}\, S) \\
= & \quad [\text{ see proposition T.8 (2) }] \\
 & \{x : x \in \mathbf{t}T \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\lceil A\}
\end{array}
$$

Furthermore:

$$
\begin{array}{ll}
 & x\lceil \mathbf{a}S \in \mathbf{t}S \ \wedge \ x \in \mathbf{t}T \\
\Rightarrow & \quad [\text{ take } y = x \text{ }] \\
 & (\exists y : y \in \mathbf{t}T \ \wedge \ y\lceil \mathbf{a}S \in \mathbf{t}S : y\lceil A = x\lceil A)
\end{array}
$$

Hence:

$$
\begin{array}{ll}
 & \mathbf{t}(T \underline{\mathbf{b}} (T \underline{\mathbf{b}} S)) \\
= & \quad [\text{ see proposition T.8 (2) }] \\
 & \{x : x \in \mathbf{t}T \ \wedge \ x\lceil A \in \mathbf{t}(T \underline{\mathbf{b}} S) : x\lceil \mathbf{a}S\} \\
= & \quad [\text{ equality above }] \\
 & \{x : x \in \mathbf{t}T \ \wedge \ (\exists y : y \in \mathbf{t}T \ \wedge \ y\lceil \mathbf{a}S \in \mathbf{t}S : y\lceil A = x\lceil A) : x\lceil \mathbf{a}S\} \\
\supseteq & \quad [\text{ implication above }] \\
 & \{x : x \in \mathbf{t}T \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\lceil \mathbf{a}S\} \\
= & \quad [\ S \subseteq T\lceil \mathbf{a}S\ ] \\
 & \mathbf{t}S
\end{array}
$$

□

In general, it is not true that $T \underline{\mathbf{b}} (T \underline{\mathbf{b}} S) = S$ as the following example shows.

Example T.12

$$
\begin{array}{l}
T = \langle \{ac, ad\}, \{a, c, d\} \rangle \\
S = \langle \{c\}, \{c, d\} \rangle
\end{array}
$$

Then we have:

$$
\begin{array}{ll}
 & \mathbf{t}(T \underline{\mathbf{b}} (T \underline{\mathbf{b}} S)) \\
= & \\
 & \mathbf{t}(T \underline{\mathbf{b}} \langle \{a\}, \{a\} \rangle) \\
= & \\
 & \{c, d\} \\
\neq & \\
 & \mathbf{t}S
\end{array}
$$

The inequality is due to the fact that we can find x and y (both $\in \mathbf{t}T$) with:

$$x\lceil (\mathbf{a}T \backslash \mathbf{a}S) = y\lceil (\mathbf{a}T \backslash \mathbf{a}S) \ \wedge \ y\lceil \mathbf{a}S \in \mathbf{t}S \ \wedge \ x\lceil \mathbf{a}S \notin \mathbf{t}S$$

Here $x = ad$ and $y = ac$. □

T.5 Other operators on trace structures

Definition T.13 *Given two trace structures T and S, then we define the* union *of T and Q, denoted $T \cup S$ (pronounce "T or S"), by*

$$\langle \mathbf{a}T \cup \mathbf{a}S, \mathbf{t}T \cup \mathbf{t}S \rangle$$

We define the intersection *of T and S, denoted $T \cap S$ (pronounce "T and S"), by*

$$\langle \mathbf{a}T \cap \mathbf{a}S, \mathbf{t}T \cap \mathbf{t}S \rangle$$

And, if $\mathbf{a}T = \mathbf{a}S$, the exclusion *of T and S, denoted $T \backslash S$ (pronounce "T without S"),*

by

$$\langle \mathbf{a}T, \mathbf{t}T \backslash \mathbf{t}S \rangle$$

□

Property T.14 *For trace structures T and S, we have:*

(1) $\mathbf{a}T = \mathbf{a}S \;\Rightarrow\; (T \underline{\mathbf{w}}\; S) = T \cap S$
(2) $\mathbf{a}S \subseteq \mathbf{a}T \;\Rightarrow\; (T \underline{\mathbf{w}}\; S)\lceil \mathbf{a}S = (T\lceil \mathbf{a}S) \cap S$

proof: Part (1): see property 1.20 in [Sne85].
Part (2):

$x \in \mathbf{t}(T\lceil \mathbf{a}S \cap S)$
$\Leftrightarrow$
$x \in \mathbf{t}T\lceil \mathbf{a}S \;\wedge\; x \in \mathbf{t}S$
$\Leftrightarrow$ [definition of [ ]
$(\exists y : y \in \mathbf{t}T : y\lceil \mathbf{a}S = x) \;\wedge\; x \in \mathbf{t}S$
$\Leftrightarrow$
$(\exists y : y \in \mathbf{t}T \;\wedge\; y\lceil \mathbf{a}S \in \mathbf{t}S : x = y\lceil \mathbf{a}S)$
$\Leftrightarrow$ [definition of $\underline{\mathbf{w}}$]
$(\exists y : y \in \mathbf{t}(T \underline{\mathbf{w}}\; S) : x = y\lceil \mathbf{a}S)$
$\Leftrightarrow$ [definition of [ ]
$x \in (T \underline{\mathbf{w}}\; S)\lceil \mathbf{a}S$

□

Property T.15 *For trace structures T and U and alphabet A, we have:*

(1) $(T \underline{\mathbf{w}}\; U)\lceil A \;\subseteq\; T\lceil A \underline{\mathbf{w}}\; U\lceil A$
(2) $\mathbf{a}T \cap \mathbf{a}U \subseteq A \;\Rightarrow\; (T \underline{\mathbf{w}}\; U)\lceil A = T\lceil A \underline{\mathbf{w}}\; U\lceil A$
(3) $(T \underline{\mathbf{b}}\; U)\lceil A \;\subseteq\; T\lceil A \underline{\mathbf{b}}\; U\lceil A$
(4) $\mathbf{a}T \cap \mathbf{a}U \subseteq A \;\Rightarrow\; (T \underline{\mathbf{b}}\; U)\lceil A = U\lceil A \underline{\mathbf{b}}\; U\lceil A$

proof: See properties 1.15, 1.16, and 1.31 in [Sne85]. □

Corollary T.16

$$(T_1 \underline{\mathbf{w}}\; T_2)\lceil \mathbf{a}T_1 \subseteq T_1$$

proof:

$(T_1 \underline{\mathbf{w}}\; T_2)\lceil \mathbf{a}T_1$
$\subseteq$ [property T.15 (1)]
$T_1\lceil \mathbf{a}T_1 \underline{\mathbf{w}}\; T_2\lceil \mathbf{a}T_1$
$=$ [property T.14 (1)]
$T_1 \cap T_2\lceil \mathbf{a}T_1$
$\subseteq$
T_1

□

Property T.17 *For trace structures T, S, and U, with $\mathbf{a}T = \mathbf{a}S$, we have:*

$$
\begin{array}{lll}
(1) & U \underline{\mathbf{w}} (T \cup S) & = (U \underline{\mathbf{w}} T) \cup (U \underline{\mathbf{w}} T) \\
(2) & U \underline{\mathbf{w}} (T \cap S) & = (U \underline{\mathbf{w}} T) \cap (U \underline{\mathbf{w}} T) \\
(3) & U \underline{\mathbf{b}} (T \cup S) & = (U \underline{\mathbf{b}} T) \cup (U \underline{\mathbf{b}} S) \\
(4) & U \underline{\mathbf{b}} (T \cap S) & \subseteq (U \underline{\mathbf{b}} T) \cap (U \underline{\mathbf{b}} S) \\
(5) & U \underline{\mathbf{w}} (T \backslash S) & = (U \underline{\mathbf{w}} T) \backslash (U \underline{\mathbf{w}} S) \\
(6) & U \underline{\mathbf{b}} (T \backslash S) & = (U \underline{\mathbf{b}} T) \backslash (U \underline{\mathbf{b}} S)
\end{array}
$$

proof: Parts (1) to (4): see properties 1.21 and 1.34 in [Sne85].
Part (5):

$$
\begin{array}{ll}
 & U \underline{\mathbf{w}} (T \backslash S) \\
= & \quad [\text{ definition of } \backslash \text{ and } \mathbf{a}T = \mathbf{a}S \text{ }] \\
 & U \underline{\mathbf{w}} \langle \mathbf{t}T \backslash \mathbf{t}S, \mathbf{a}T \rangle \\
= & \quad [\text{ definition of } \underline{\mathbf{w}} \text{ }] \\
 & \langle \mathbf{a}T \cup \mathbf{a}U \\
 & , \{x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \ \wedge \ x\lceil \mathbf{a}U \in \mathbf{t}U \ \wedge \ x\lceil \mathbf{a}T \in \mathbf{t}T \backslash \mathbf{t}S : x\} \\
 & \rangle \\
= & \quad [\text{ } \mathbf{a}T = \mathbf{a}S \text{ }] \\
 & \langle \mathbf{a}T \cup \mathbf{a}U \\
 & , \{x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \ \wedge \ x\lceil \mathbf{a}U \in \mathbf{t}U \ \wedge \ x\lceil \mathbf{a}T \in \mathbf{t}T : x\} \backslash\backslash \\
 & \quad \{x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \ \wedge \ x\lceil \mathbf{a}U \in \mathbf{t}U \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\} \\
 & \rangle \\
= & \quad [\text{ definition of } \backslash \text{ and } \mathbf{a}T = \mathbf{a}S \text{ }] \\
 & \langle \mathbf{a}T \cup \mathbf{a}U, \{x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \ \wedge \ x\lceil \mathbf{a}U \in \mathbf{t}U \ \wedge \ x\lceil \mathbf{a}T \in \mathbf{t}T : x\} \backslash\backslash \\
 & \langle \mathbf{a}S \cup \mathbf{a}U, \{x : x \in (\mathbf{a}T \cup \mathbf{a}U)^* \ \wedge \ x\lceil \mathbf{a}U \in \mathbf{t}U \ \wedge \ x\lceil \mathbf{a}S \in \mathbf{t}S : x\} \rangle \\
= & \quad [\text{ definition of } \backslash \text{ }] \\
 & (U \underline{\mathbf{w}} T) \backslash (U \underline{\mathbf{w}} S)
\end{array}
$$

Part (6) is similar. □

Property T.18 *For trace structures T and U and alphabet A, we have:*

$$
\begin{array}{lll}
(1) & (T\lceil A) \backslash (U\lceil A) & \subseteq (T \backslash U)\lceil A \\
(2) & (T \cup U)\lceil A & = (T\lceil A) \cup (U\lceil A) \\
(3) & (T \cap U)\lceil A & \subseteq (T\lceil A) \cap (U\lceil A)
\end{array}
$$

proof: Part (1):

$$
\begin{array}{ll}
 & x \in \mathbf{t}((T\lceil A) \backslash (U\lceil A)) \\
\Leftrightarrow & \quad [\text{ definition of } \backslash \text{ and } \lceil \text{ }] \\
 & (\exists y : y \in \mathbf{t}T : y\lceil A = x) \ \wedge \ (\forall y : y \in \mathbf{t}U : y\lceil A \neq x) \\
\Leftrightarrow & \\
 & (\exists y : y \in \mathbf{t}T : y\lceil A = x) \ \wedge \ (\forall y : y\lceil A = x : y \notin \mathbf{t}U) \\
\Rightarrow & \\
 & (\exists y : y \in \mathbf{t}T \ \wedge \ y \notin \mathbf{t}U : y\lceil A = x) \\
\Leftrightarrow & \quad [\text{ definition of } \backslash \text{ and } \lceil \text{ }] \\
 & x \in \mathbf{t}(T \backslash U)\lceil A
\end{array}
$$

Part (2):

$$
\begin{array}{ll}
 & x \in \mathbf{t}(T \cup U)\lceil A \\
\Leftrightarrow & \text{[definition of } \lceil \text{]} \\
 & (\exists y : y \in \mathbf{t}(T \cup U) : y\lceil A = x) \\
\Leftrightarrow & \text{[definition of } \cup \text{]} \\
 & (\exists y : y \in \mathbf{t}T : y\lceil A = x) \ \vee \ (\exists y : y \in \mathbf{t}U : y\lceil A = x) \\
\Leftrightarrow & \text{[definition of } \cup \text{ and } \lceil \text{]} \\
 & x \in \mathbf{t}(T\lceil A) \cup \mathbf{t}(T\lceil A)
\end{array}
$$

Part (3):

$$
\begin{array}{ll}
 & x \in \mathbf{t}(T \cap U)\lceil A \\
\Leftrightarrow & \text{[definition of } \lceil \text{]} \\
 & (\exists y : y \in \mathbf{t}(T \cap U) : y\lceil A = x) \\
\Rightarrow & \text{[definition of } \cap \text{]} \\
 & (\exists y : y \in \mathbf{t}T : y\lceil A = x) \ \wedge \ (\exists y : y \in \mathbf{t}U : y\lceil A = x) \\
\Leftrightarrow & \text{[definition of } \cap \text{ and } \lceil \text{]} \\
 & x \in \mathbf{t}(T\lceil A) \cap \mathbf{t}(T\lceil A)
\end{array}
$$

□

Definition T.19 *The* prefix closure *of a trace set S is defined by:*

$$\mathbf{pref}(S) = \{x : (\exists z :: xz \in S) : x\}$$

and the prefix closure of a trace structure T by:

$$\mathbf{pref}(T) = \langle \mathbf{pref}(\mathbf{t}T), \mathbf{a}T \rangle$$

□

We have $\mathbf{pref}(\mathbf{t}T) = \mathbf{t}(\mathbf{pref}(T))$. For cosmetic reasons we prefer to write $\mathbf{pref}(\mathbf{t}T)$.

Property T.20

(1) $\mathbf{pref}(T \ \underline{\mathbf{w}} \ S) \subseteq \mathbf{pref}(T) \ \underline{\mathbf{w}} \ \mathbf{pref}(S)$
(2) $\mathbf{a}T \cap \mathbf{a}U = \emptyset \Rightarrow \mathbf{pref}(T \ \underline{\mathbf{w}} \ S) = \mathbf{pref}(T) \ \underline{\mathbf{w}} \ \mathbf{pref}(S)$
(3) $\mathbf{pref}(T \ \underline{\mathbf{b}} \ S) \subseteq \mathbf{pref}(T) \ \underline{\mathbf{b}} \ \mathbf{pref}(S)$
(4) $\mathbf{a}T \cap \mathbf{a}U = \emptyset \Rightarrow \mathbf{pref}(T \ \underline{\mathbf{b}} \ S) \subseteq \mathbf{pref}(T)\mathbf{bpref}(S)$

proof: See [Sne85], properties 1.18, 1.19, and 1.33 □

Property T.21 *Weaving of prefix closed structures is prefix closed.*
proof: See [Sne85] property 1.17 □

Appendix S
Supervisory theory

In this section we discuss the supervisory control theory of Ramadge and Wonham (see [RaWo]).[1] In this theory a discrete system is viewed as a sequential process that, in fact, is equal to a (possibly infinite) state graph. Each label in such a graph represents the occurrence of some event. The labels may be controlled in which case they can be enabled or disabled. A supervisor is a second graph that observes the behaviour of the first one and enables or disables the controlled events in order to get a predefined behaviour. Formally the definitions are as follows:

Definition S.1 *A* sequential process *is a tuple* $G = (Q, \Sigma, \delta, q_0, Q_m)$ *with*

Q	*set of states (need not be finite)*
Σ	*finite set of events*
$q_0 \in Q$	*initial state*
$Q_m \subseteq Q$	*marker states*
$\delta : Q \times \Sigma \rightarrow Q$	*(partial) transition function*

We are also given a subset $\Sigma_c \subseteq \Sigma$, *denoting the set of* controlled events *and* $\Sigma_u = \Sigma \backslash \Sigma_c$, *denoting the set of* uncontrolled events.
A set of control patterns *is* $\Gamma = \{\gamma : (\gamma : \Sigma \rightarrow \{\mathbf{true}, \mathbf{false}\}) \ \wedge \ \gamma(\Sigma_u) : \gamma\}$ *and we say* $\sigma \in \Sigma$ *is enabled if* $\gamma(\sigma)$ *and is disabled if* $\neg\gamma(\sigma)$.
A controlled sequential process *is a tuple* $\mathcal{G} = (G, \Gamma)$.
An extended control grammar *is* $G_c = (Q, \Gamma \times \Sigma, \delta_c, q_0, Q_m)$ *with*[2]

$$\begin{array}{rcll} \delta_c(q,(\gamma,\sigma)) & = & \delta(q,\sigma) & \textit{if} \quad \gamma(\sigma) \\ & = & \perp & \textit{if} \quad \neg\gamma(\sigma) \end{array}$$

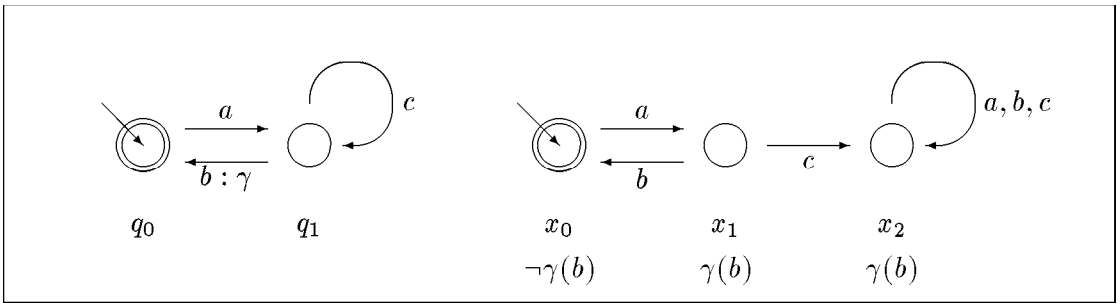
□

G is in fact a state graph representation of a trace structure with trace set equal to the set of all paths in G starting in q_0 and ending in a marker state. The only difference with state graphs is that δ need not be complete in G. It is straightforward, however, to create a complete graph out of G: the symbol $\perp$ represents the error state if we add:

$$\delta(\perp, \sigma) = \perp \qquad \delta_c(\perp, \gamma, \sigma) = \perp$$

[1] In fact, we are inspired by the work of Ramadge and Wonham and have adapted their problem formulation in this report.

[2] $\perp$ stands for undefined.



Figuur S.1: Sequential process G (left) and supervisor S (right).

G_c represents the behaviour of G under the control pattern Γ (i.e., all disabled events are removed from the graph G to get the graph G_c). G_c is in fact again a state graph where transitions labeled with events that are disabled are removed (i.e., replaced by a transition to the error state).

Definition S.2 *A* supervisor *is a tuple* $\mathcal{S} = (S, \phi)$ *with*

$S = (X, \Sigma, \xi, x_0, X_m)$ *another sequential process*
$\phi : X \rightarrow \Gamma$ *the state feedback map*

The closed loop supervised process *can then be defined as*

$$\mathcal{S}|\mathcal{G} = \mathbf{re}(X \times Q, \Sigma, \psi, (x_0, q_0), X_m \times Q_m)$$

with

$$\begin{array}{rcll} \psi(x,q,\sigma) & = & (\xi(x,\sigma), \delta(q,\sigma)) & \text{if } \phi(x)(\sigma) \wedge \xi(x,\sigma) \neq \perp \wedge \delta(q,\sigma) \neq \perp \\ & = & \perp & \text{otherwise} \end{array}$$

and **re** *meaning: the reachable part of the sequential process (graph) only.* □

$\mathcal{S}|\mathcal{G}$ is the cartesian product of $\mathcal{S}$ and $\mathcal{G}$ with properly defined transition function and consisting of the accessible (reachable) part only. In fact, $\mathcal{S}|\mathcal{G}$ is the weave of the graphs of S and G, where enabling and disabling of events in G depends on the corresponding state of S.

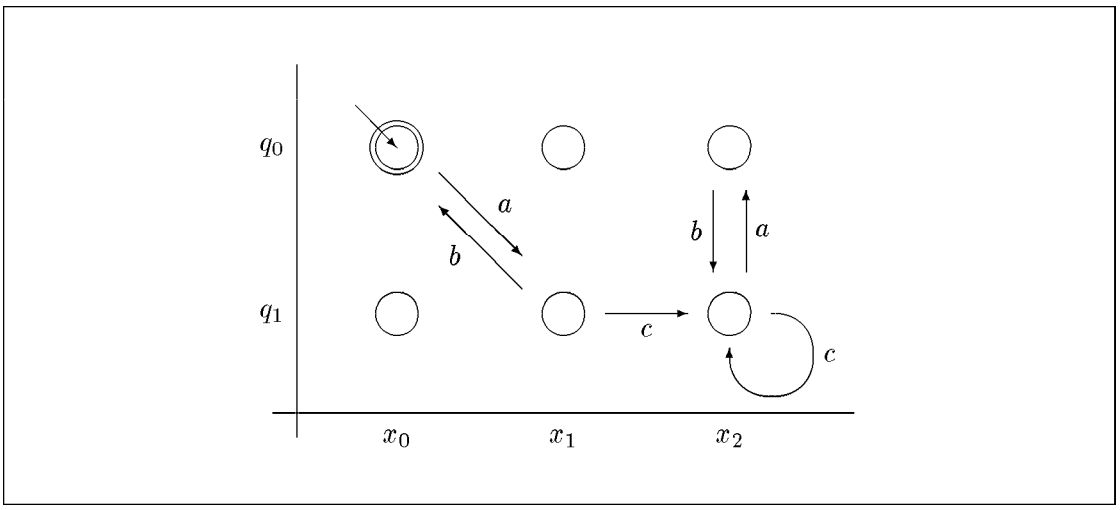
Example S.3 Consider the sequential process G and the supervisor S as given in figure S.1. We have

$$G = (\{q_0, q_1\}, \{a, b, c\}, \delta, q_0, \{q_0\})$$

with $\delta(q_0, a) = q_1$, $\delta(q_1, b) = q_0$, $\delta(q_1, c) = q_1$, and all other δ's undefined. Moreover, $\Sigma_c = \{b\}$, denoted by $b : \gamma$ in the graph. The supervisor is given by

$$S = (\{x_0, x_1, x_2\}, \{a, b, c\}, \xi, x_o, \{x_0\})$$

with ξ according to the graph and state feedback map given by $\phi(x_0) = \{\neg\gamma(b)\}$,



Figuur S.2: Corresponding closed loop supervised process for figure S.1

$\phi(x_1) = \{\gamma(b)\}$, and $\phi(x_2) = \{\gamma(b)\}$, i.e., b is disabled if the supervisor is in state x_0 and enabled if it is in state x_1 or x_2.

According to the above definitions, we find the closed loop supervised process $\mathcal{S}|\mathcal{G}$ as in figure S.2. For example $\psi(x_1, q_1, b) = (\xi(x_1, b), \delta(q_1, b)) = (x_0, q_0)$, because in x_1 we have $\gamma(b)$. If we had $\neg\gamma(b)$ in x_1, we should have $\psi(x_1, q_1, b) = \perp$.

If we choose $\neg\gamma(b)$ in x_2 of S, we disable b although we could observe it. S is overdone here: we are able to observe an event that is at that moment disabled. If we choose not to have the occurrence of b in state x_2 of S but leave $\gamma(b)$ there, we enable b but are unable to observe it. S is not complete: it cannot observe at every point every event that can occur. If S is not overdone nor incomplete, we call S proper. We only investigate proper supervisors in this section. □

These definitions have a lot in common with our definitions of a discrete system. A sequential process is defined here as a (generator) state graph with transitions that may be blocked (i.e., disabled). Each path through the graph gives a trace of the process. If such a path ends in a marker state, the corresponding trace represents a completed task. Marker states can therefore be identified with task states in the state graph corresponding with a discrete system.

A supervisor is an (observer) state graph with all events enabled. The task of the supervisor is to follow the behaviour of the sequential process and compute (after each occurrence of an event) a new control pattern so as to enable or disable future events. However, the behaviour of plant (sequential process) and controller (supervisor) are not the same. The plant is a passive generator (it generates events, but is unable to control) while the supervisor is an active observer (it cannot generate events on its own, it can only enable and disable events). Enabling an event does not mean that this event actually occurs; it is possible that another event that is also enabled actually occurs.

It turns out that this kind of control is in fact a passive kind of control: a process is not forced to do anything, only certain actions may temporarily be disabled. In CODE an active kind of control is developed.

The different interpretation of plant and supervisor in supervisory control makes it hard to give a more general definition of connection of processes for example to connect more than two processes or to supervise a supervisor. Indeed, nothing is said about the behaviour of a number of processes if they can influence each other and the observational behaviour of the supervisor cannot be influenced either.

Last, but not least, a sequential process with state graphs seems suitable but is not. As long as the graphs are finite (i.e., as long as the corresponding behaviour is regular), it is always possible to draw such a graph, but how can one draw an infinite graph? In CODE the behaviour is given in language-like terms directly and a solution is given that is independent of the properties of that behaviour. The only restriction is that one should be able to compute the necessary blends ($]\!|$) and weaves ($\|$).

In supervisory theory a number of languages is defined also:

Definition S.4 *In addition to a sequential process G and a supervisor S, we define:*

$$\begin{aligned} L_m(\mathcal{G}) &= \{s : s \in \Sigma^* \ \wedge \ \delta(q_0, s) \in Q_m : s\} \\ L(\mathcal{S}|\mathcal{G}) &= \{s : s \in \Sigma^* \ \wedge \ \psi(x_0, q_0, s) \neq \perp : s\} \\ L_c(\mathcal{S}|\mathcal{G}) &= L(\mathcal{S}|\mathcal{G}) \cap L_m(\mathcal{G}) \\ L_m(\mathcal{S}|\mathcal{G}) &= \{s : s \in \Sigma^* \ \wedge \ \psi(x_0, q_0, s) \in X_m \times Q_m\} \end{aligned}$$

□

Example S.5 In example S.3 we have:

$$\begin{aligned} L_m(\mathcal{G}) &= (ac^*b)^* \\ L(\mathcal{S}|\mathcal{G}) &= (ac^*b)^* \\ L_c(\mathcal{S}|\mathcal{G}) &= (ac^*b)^* \\ L_m(\mathcal{S}|\mathcal{G}) &= (ab)^* \end{aligned}$$

□

The main control problem in supervisory control is formulated as: given a sequential process G, minimal acceptable behaviour L_a, and a legal behaviour L_g, find a supervisor S such that:

$$L_a \subseteq L_c(\mathcal{S}|\mathcal{G}) \subseteq L_g$$

called the Supervisory control problem (SCP), or

$$L_a \subseteq L_m(\mathcal{S}|\mathcal{G}) \subseteq L_g$$

called the Supervisory marker problem (SMP).

For SCP we need the enabling and disabling of events, for SMP it is enough to find a supervisor that (in every state) enables every event, that can be accepted (i.e., the supervisors should be complete with respect to G_c). This last problem is very similar to ECODE.

For proper supervisors we can in fact identify $L_m(\mathcal{S}|\mathcal{G})$ with $P \parallel R$, where P is the discrete system corresponding to $\mathcal{G}$ and R is the controller corresponding to $\mathcal{S}$, i.e.,

$$\begin{aligned} P &= \langle \Sigma, L_m(G) \rangle \\ R &= \langle \Sigma, L_m(S) \rangle \end{aligned}$$

which gives: $\mathbf{t}(P \parallel R) = L_m(\mathcal{S}|\mathcal{G})$ so that we can rewrite SMP as $L_a \subseteq P \parallel R \subseteq L_g$ (which is ECODE).

The other way round, a ECODE-problem can be written as a SMP only if the exogenous alphabet of the system to be controlled is empty. If $\mathbf{e}P = \emptyset$ we can rewrite ECODE in SMP:

$L_a = L_{min}$
$L_g = L_{max}$
$L_m(\mathcal{S}|\mathcal{G}) = \mathbf{t}(P \parallel R)$

However, if $\mathbf{e}P \neq \emptyset$, we have that $\mathbf{a}R = \mathbf{c}P$, so $\mathbf{a}R \neq \mathbf{a}P$. We control P with only part of its events observable (and thus controllable) by R. In supervisory control this phenomenon (partial control) is treated by using masks, i.e., every event in P is mapped onto zero or one event of R (see [CDFV88]).

It turns out that the theory presented in this report has some overlap with supervisory control: SMP is equal to ECODE (with $\mathbf{e}P = \emptyset$). In general, however, supervisory control uses another approach of controlling events (by enabling and disabling). In CODE a number of events control the other events. CODE seems more general because it allows partial control as well as marker control without using any extra mathematical equipment.

Appendix A
Answers to exercises

1.1 $(a)\ a^2b^3 = aabbb$
$(b)\ (a^*b)^2 = \{bb, abab, aabab, abaab, aabaab, \ldots\}$
$(c)\ (a^*b^*)^* =$ all combinations of a and b
$(d)\ ab, bc = abc$
$(e)\ ab, ba =$ undefined
$(f)\ ab, c = \{cab, acb, abc\}$
$(g)\ abaab\lceil\{a\} = aaa$
$(h)\ abaab\lceil\{c\} = \epsilon$
$(i)\ \mathbf{pref}(aba) = \{\epsilon, a, ab, aba\}$
$(j)\ \mathbf{pref}(a^*ba^*) =$ all combinations of a and b with at most one b

1.2 P is lock free
$\Rightarrow\ \mathbf{b}P = \mathbf{pref}(\mathbf{t}P)$
$\Rightarrow\ [\ \mathbf{pref}(\mathbf{b}P) = \mathbf{pref}(\mathbf{pref}(\mathbf{t}P)) = \mathbf{pref}(\mathbf{t}P) = \mathbf{b}P\ \wedge\ \mathbf{t}P \subseteq \mathbf{pref}(\mathbf{t}P)\ ]$
$\mathbf{b}P = \mathbf{pref}(\mathbf{b}P\ \wedge\ \mathbf{t}P \subseteq \mathbf{b}P$
$\Rightarrow\ P$ is realistic

1.3 First $\subseteq$: $\emptyset \subseteq \mathbf{b}P\ \wedge\ \emptyset \subseteq \mathbf{t}P$
Second $\subseteq$: $\mathbf{b}P \subseteq (\mathbf{a}P)^*\ \wedge\ \mathbf{t}P \subseteq (\mathbf{a}P)^*$

1.4 $P_1 \subseteq P_2$ is not defined. $P_3 \subseteq P_4$ is true.
Realistic are P_1 and P_2
Lock free is P_2.

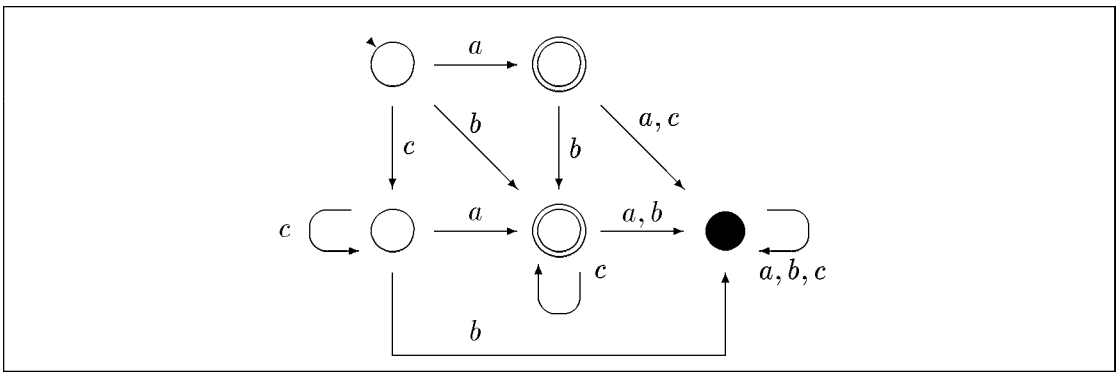
1.5 $\langle\{a, b\}, (\epsilon|a), (a)\rangle$

1.6 $\mathbf{real}(\sim(\langle\{a, b\}, \mathbf{pref}(a|b), (a|b)\rangle)) = \mathbf{real}(\langle\{a, b\}, (a|b)(a|b)^+, \epsilon|(a|b)(a|b)^+\rangle)$
$= \mathbf{empty}(\{a, b\})$
$\mathbf{real}(\sim(\langle\{a\}, (a|aa), (a)\rangle)) = \mathbf{real}(\langle\{a\}, (\epsilon|a^3a^+), (\epsilon|aa^+)\rangle) = \mathbf{skip}(\{a\})$

1.7 $\langle\{a, b, c\}, \mathbf{pref}(abc), \epsilon)\rangle$
$\mathbf{empty}(\{a, b, c\})$
$\langle\{a, b, c\}, (c|ca|cab|ac|acb|abc), (cab|acb|abc)\rangle$

1.8 $\langle\{c\}, \mathbf{pref}(c), \epsilon)\rangle$
$\mathbf{empty}(\{a, b\})$
$\langle\{a, b, c\}, (c|ca|cab|ac|acb|abc), (cab|acb|abc)\rangle$

1.9 $\langle\{a, b, c\}, \mathbf{pref}(a^*|ab), (a|ab)\rangle$
$\langle\{a, b, c\}, \mathbf{pref}(a), \emptyset\rangle$
$\langle\{a, b, c\}, (ab), \emptyset\rangle$



Figuur A.1: State graph for system P from exercise 2.1

1.10 Take for example $P = \langle\{a,b,c\},(ab|ac)\rangle$ and $A_1 = \{a,b\}$ and $A_2 = \{b,c\}$. Then $P\lceil A_1 \parallel P\lceil A_2 = \langle\{a,b,c\},(ab|cab|acb|abc|ac|ca)\rangle \neq P$.

1.11
$$\begin{aligned} & x \in \mathbf{b}(P \parallel R) \\ = \;& x\lceil \mathbf{a}P \in \mathbf{b}P \;\wedge\; x\lceil \mathbf{a}R \in \mathbf{b}R \\ = \;& x\lceil \mathbf{a}P \in \mathbf{b}P \;\wedge\; x\lceil A \in A^* \;\wedge\; x\lceil \mathbf{a}R \in \mathbf{b}R \;\wedge\; x\lceil A \in A^* \\ = \;& x \in \mathbf{b}(P \parallel A^*) \;\wedge\; x \in \mathbf{b}(R \parallel A^*) \\ = \;& x \in \mathbf{b}((P \parallel A^*) \cap (R \parallel A^*)) \end{aligned}$$
and similar for $\mathbf{t}$.

1.12 $((ba|\epsilon) \parallel (ab|\epsilon)) \parallel (ab|\epsilon) = (\epsilon) \parallel (ab|\epsilon) = (ab|\epsilon)$
$(ba|\epsilon) \parallel ((ab|\epsilon) \parallel (ab|\epsilon)) = ((ba|\epsilon) \parallel (ab|\epsilon) = (\epsilon)$

1.13 $P!? = \langle\{a!,b!,c?\},(a!|b!|c?)^*\rangle \quad R!? = \langle\{a?,b?,c!\},(a? \cdot b? \cdot c!)\rangle$
$\mathbf{t}(\mathbf{trans}\{a,b,c\}) = ((a! \cdot a?),(b! \cdot b?),(c! \cdot c?))$
$\mathbf{t}(P \overleftrightarrow{\parallel} R) = ((a! \cdot a? \cdot b! \cdot b? \mid a! \cdot b! \cdot a? \cdot b? \mid a! \cdot b! \cdot b? \cdot a?) \cdot c! \cdot c?)$

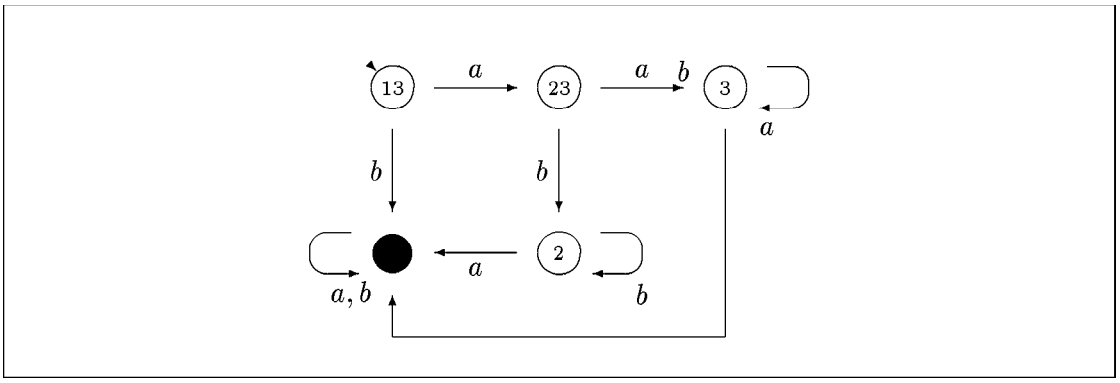
2.1 $\mathbf{sg}(P)$ can be found in figure A.1. The DES corresponding to the state graph of figure A.1 equals $\langle\{a,b,c\},(a|b|bc),(a|bc)\rangle$

2.2 $\mathbf{des}(G_{\mathrm{nd}}) = \langle\{a,b\},(a^*|ab^*\rangle$.
To get a deterministic graph we compute:

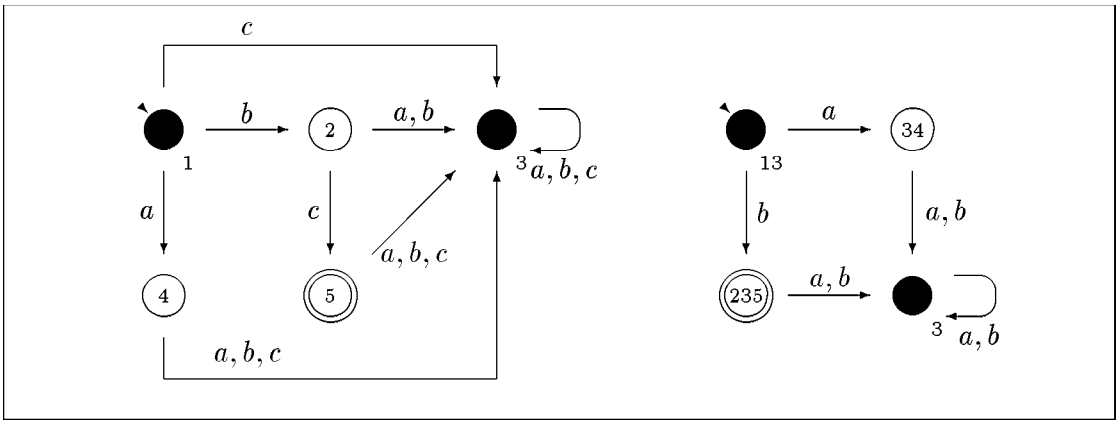
$$\begin{array}{lcl} \gamma^*(q,\epsilon) & = & \{p_1,p_3\} \;=\; \overline{q} \\ \gamma^*(\{p_1,p_3\},a) & = & \{p_2,p_3\} \\ \gamma^*(\{p_1,p_3\},b) & = & \emptyset \\ \gamma^*(\{p_2,p_3\},a) & = & \{p_3\} \\ \gamma^*(\{p_2,p_3\},b) & = & \{p_2\} \\ \gamma^*(\{p_2\},a) & = & \emptyset \\ \gamma^*(\{p_2\},b) & = & \{p_2\} \\ \gamma^*(\{p_3\},a) & = & \{p_3\} \\ \gamma^*(\{p_3\},b) & = & \emptyset \\ \gamma^*(\emptyset,a) & = & \emptyset \\ \gamma^*(\emptyset,b) & = & \emptyset \end{array}$$

which leads to the graph of figure A.2.

2.3 The state graphs can be found in figure A.3. The nd-graph for $P\lceil\{a,b\}$ is equal to that of P, but with all c changed to ϵ.



Figuur A.2: Deterministic graph for the graph of figure 2.5



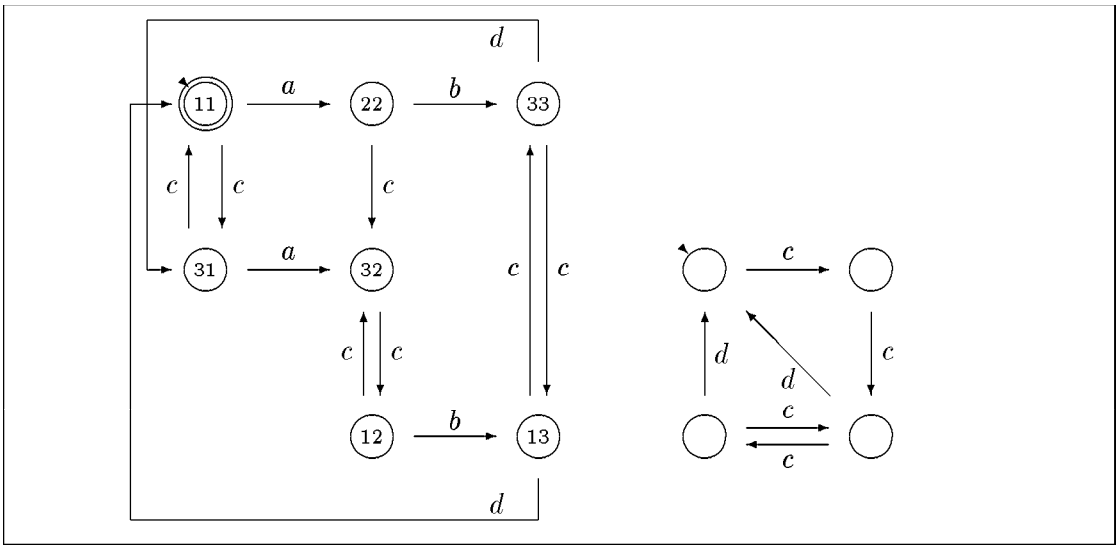
Figuur A.3: State graphs for exercise 2.3: $\mathbf{sg}(P)$ *(left), and* $\mathbf{det}(\mathbf{sg}(P\lceil\{a,b\}))$ *(right). Dump states are included here.*

2.4 $G_1 \parallel G_2$ can be found in figure A.4. $G_1 \rceil\lceil G_2$ can be also found in figure A.4 and its deterministic equivalence in figure A.4 (right). Computations are similar as previous exercises.

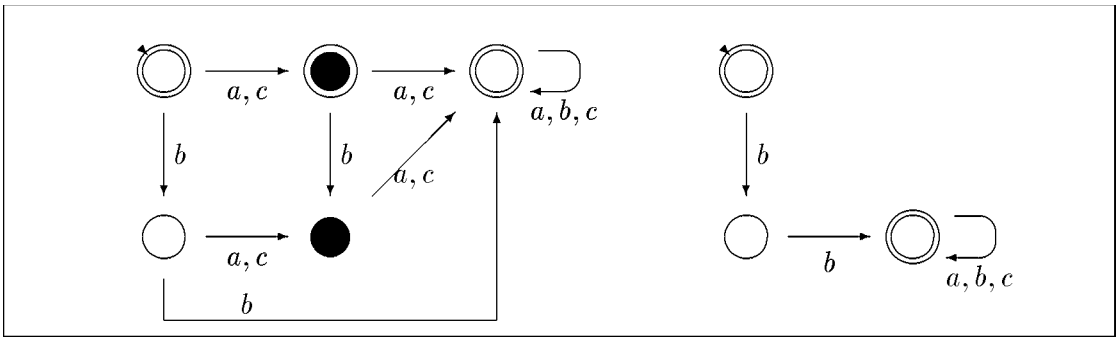
2.5 $\sim G$ and $\mathbf{real}(\sim G)$ can be found in figure A.5.

2.6 $G_1 \cup G_2$ can be found in figure A.6 (left). All arrows to state 55 are not drawn, but replaced by an arrow without label.
To find the minimal graph we use an algorithm similar to the standard algorithms for automatons. First we look at all combinations of two states an see if these state may be equivalent. If not so, we mark the corresponding place in the table:

22	X					
33	.	X				
44	X	.	X			
42	X	.	X	.		
54	X	.	X	.	.	
55	X	X	X	X	X	X
	11	22	33	44	42	54



Figuur A.4: Left: $G_1 \,||\, G_2$ for exercise 2.4 (without dump states). Right: **det**$(G_1 \,]\![\, G_2)$ *(without dump states). State graph $G_1 \,]\![\, G_2$ is equal to the left graph, but with a and b changed into ϵ.*



Figuur A.5: $\sim G$ and **real**$(\sim G)$ *for exercise 2.5*

For the remaining pairs we see if, for each alphabet element, the resulting pair perhaps is not equivalent:

$(11,33) \xrightarrow{a} (22,55)$ which are non-equivalent

$(22,44) \xrightarrow{a} (55,55)$ gives no result

$\xrightarrow{b} (44,54) \xrightarrow{a} (55,55)$

$\xrightarrow{b} (55,55)$

$\xrightarrow{c} (55,55)$ So 44 and 54 equivalent

$\xrightarrow{c} (55,55)$ So 22 and 42 equivalent

$(22,54) \xrightarrow{a} (55,55)$

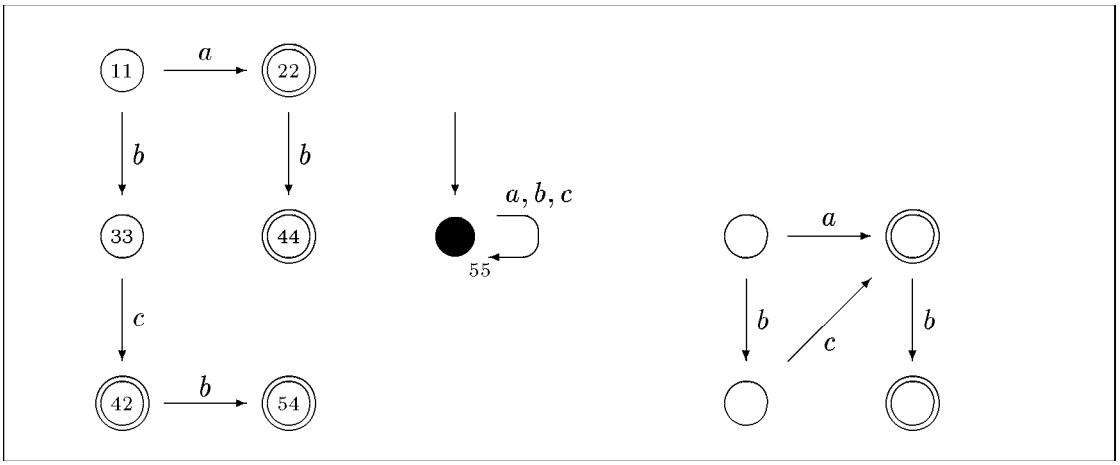
$\xrightarrow{b} (44,55)$ which are non-equivalent

$(42,54) \xrightarrow{a} (55,55)$

$\xrightarrow{b} (54,55)$ which are non-equivalent

$(44,42) \xrightarrow{a} (55,55)$

$\xrightarrow{b} (55,54)$ which are non-equivalent



Figuur A.6: $G_1 \cup G_2$ for exercise 2.6 (left) and its minimal graph (right)

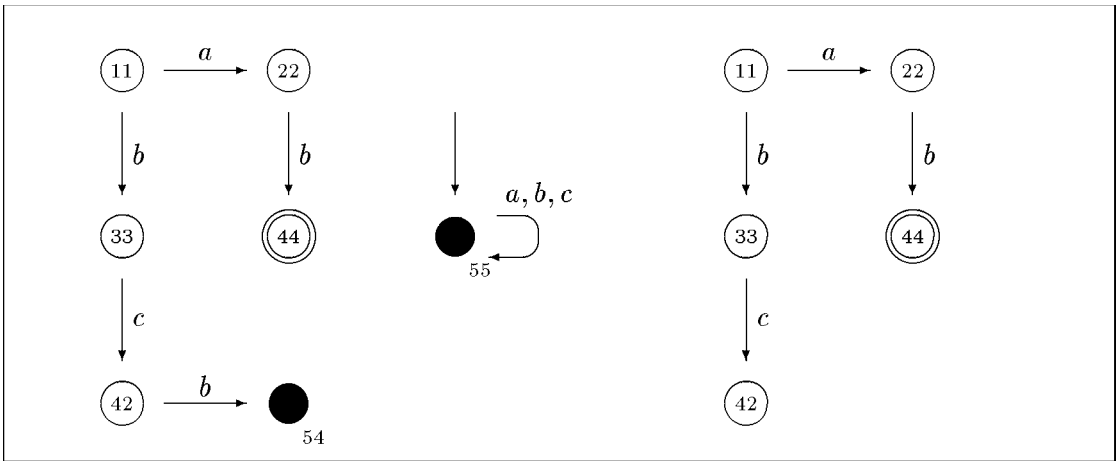
So equivalent pairs are: (11,33), (22,44), (22,54), (44,43), and (42,54). Equivalent states can be replaced by one state, which leads to the graph in the figure A.6 (right).
$G_1 \cap G_2$ can be found in figure A.7 (left). All arrows to state 55 are not drawn, but replaced by an arrow without label.
Computing the minimal graph now results in:

22	.					
33	.	.				
44	X	X	X			
42	.	.	.	X		
54	X	X	X	X	X	
55	X	X	X	X	X	.
	11	22	33	44	42	54

For the remaining pairs we see if, for each alphabet element, the resulting pair is non-equivalent:

$(11,22) \xrightarrow{a} (22,55)$ which are non-equivalent
$(11,33) \xrightarrow{a} (22,55)$ which are non-equivalent
$(11,42) \xrightarrow{a} (22,55)$ which are non-equivalent
$(22,33) \xrightarrow{a} (55,55)$
$\xrightarrow{b} (44,55)$ which are non-equivalent
$(22,42) \xrightarrow{a} (44,55)$ which are non-equivalent
$(33,42) \xrightarrow{a} (55,55)$
$\xrightarrow{b} (55,54) \xrightarrow{a} (55,55)$
$\xrightarrow{b} (55,55)$
$\xrightarrow{c} (55,55)$ So (55,54) are equivalent
$\xrightarrow{c} (42,55)$ which are non-equivalent

So equivalent pair is: (42,55). The minimal graph can be found in figure A.7 (right).



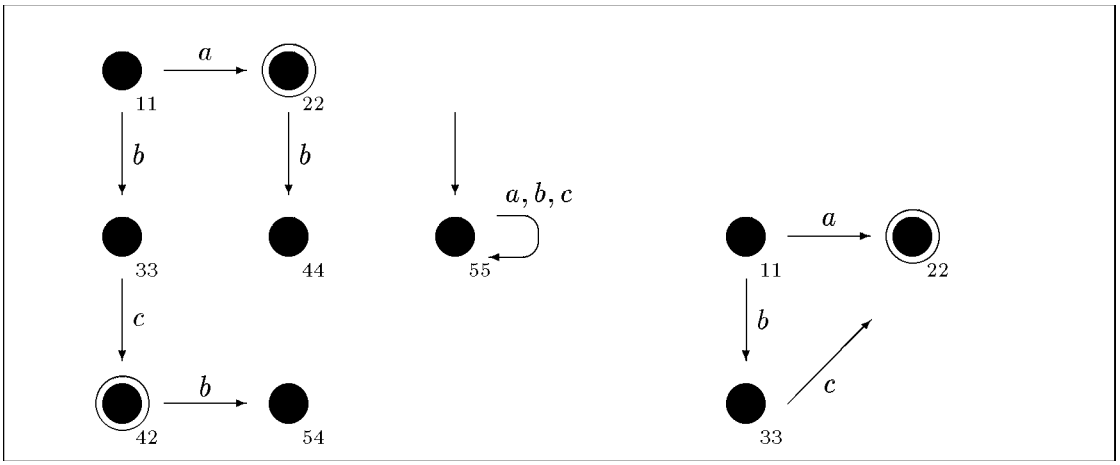
Figuur A.7: $G_1 \cup G_2$ for exercise 2.6 (left) and its minimal graph (right)

$G_1 \backslash G_2$ can be found in figure A.8 (left). All arrows to state 55 are not drawn, but replaced by an arrow without label.
Computing the minimal graph now results in:

22	X					
33	.	X				
44	.	X	.			
42	X	.	X	X		
54	.	X	.	.	X	
55	.	X	.	.	X	.
	11	22	33	44	42	54

For the remaining pairs we see if, for each alphabet element, the resulting pair is non-equivalent:

$(11,33) \xrightarrow{a} (22,42) \xrightarrow{a} (55,55)$
$\xrightarrow{b} (44,54) \xrightarrow{a} (55,55)$
$\xrightarrow{b} (55,55)$
$\xrightarrow{c} (55,55)$ ((44,54) equivalent
$\xrightarrow{c} (55,55)$ (22,42) equivalent
$\xrightarrow{b} (33,55)$ no equivalence
$(11,44) \xrightarrow{a} (22,55)$ no equivalence
$(11,54) \xrightarrow{a} (22,55)$ no equivalence
$(11,55) \xrightarrow{a} (22,55)$ no equivalence
$(33,44) \xrightarrow{a} (55,55)$
$\xrightarrow{b} (55,55)$
$\xrightarrow{c} (42,55)$ no equivalence
$(33,54) \xrightarrow{a} (55,55)$
$\xrightarrow{b} (55,55)$
$\xrightarrow{c} (42,55)$ no equivalence



Figuur A.8: $G_1 \backslash G_2$ for exercise 2.6 (left) and its minimal graph (right)

$$
\begin{array}{lll}
(33,55) \xrightarrow{a} (55,55) & \\
\xrightarrow{b} (55,55) & \\
\xrightarrow{c} (42,55) & \text{no equivalence} \\
(44,55) \xrightarrow{a} (55,55) & \\
\xrightarrow{b} (55,55) & \\
\xrightarrow{c} (42,55) & \text{no equivalence} \\
(54,55) \xrightarrow{a} (55,55) & \\
\xrightarrow{b} (55,55) & \\
\xrightarrow{c} (55,55) & \text{(54,55) equivalent}
\end{array}
$$

So equivalent pairs are: (22,42), (44,54), (44,55), and (54,55). The minimal graph can be found in figure A.8 (right).

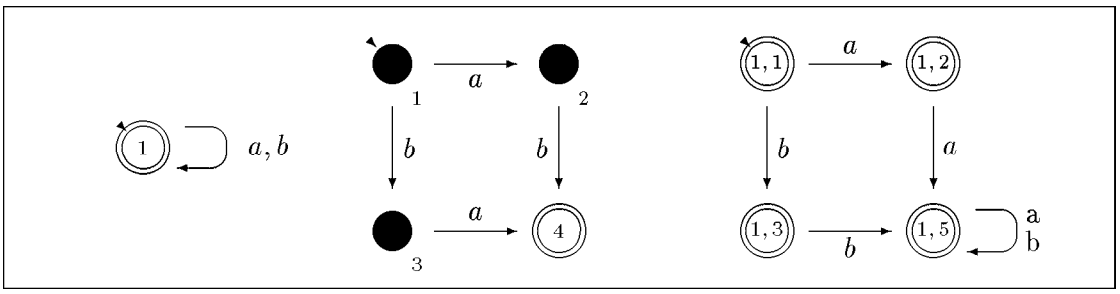
3.1 (a) $\langle\{a,b\},(ab)\rangle \| \langle\{a,b\},(ba)\rangle = \langle\{a,b\},\epsilon,\emptyset\rangle$ which is not free of deadlock: $x = \epsilon$ is a deadlock ending behaviour.
(b) $\langle\{a,b\},(ab)^*\rangle \parallel \langle\{a,b\},(aba)^*\rangle = \langle\{a,b\},\mathbf{pref}(aba),\emptyset\rangle$ which is again not deadlock free.
(c) $\langle\{a,b\},(aba)\rangle \parallel \langle\{a\},a^*\rangle = \langle\{a,b\},aba\rangle$ which is lock free.

3.2 The greatest lock free subsystems simply are:

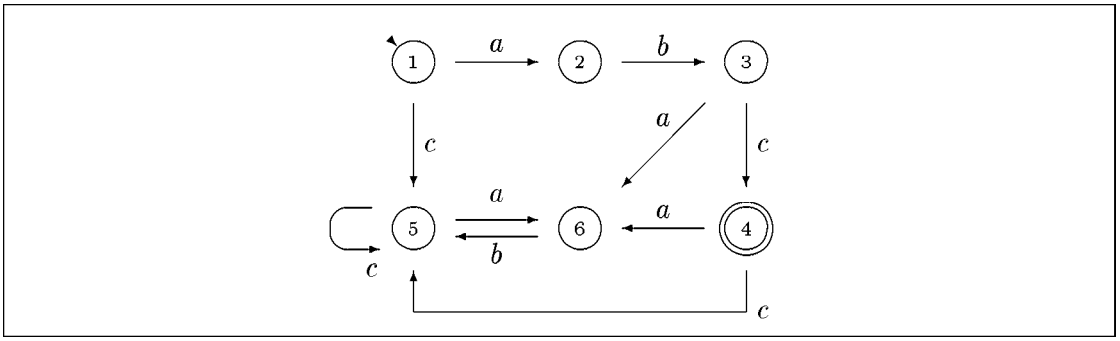
(a) $\langle\{a,b,c\},(ab)\rangle$
(b) $\langle\{a,b,c\},(ab)\rangle$
(c) $\langle\{a,b,c\},(abc)\rangle$

meaning that the lock-sets are respectively:

(a) $\mathbf{pref}(ab)^* \backslash \mathbf{pref}(ab)$
(b) $\mathbf{pref}(ab|c)^* \backslash \mathbf{pref}(ab)$
(c) $\mathbf{pref}(ab|c)^* \backslash \mathbf{pref}(abc)$



Figuur A.9: $\mathbf{sg}(P)$ *(left),* $\mathbf{sg}(T)$ *(middle), and* $\mathbf{sg}(P)\backslash\mathbf{sg}(T)$ *(right) for exercise 4.2. In the middle graph dump state (with number 5) is not drawn.*



Figuur A.10: $\mathbf{sg}(P)$ *for exercise 4.2*

3.3 In all cases we start with $R_0 = \langle \mathbf{a}R, (\mathbf{a}R)^* \rangle$. So $P \parallel R_0 = P$. We write in full $\mathbf{b}P = \{\epsilon, a, c, ab\}$. So: $\mathbf{lock}(P \parallel R_0) = \mathbf{lock}(P) = \{c\}$ in all three cases.
(a) Computations give:

$$\begin{array}{lcl} R_0 & = & \langle \{a,b\}, (a|b)^* \rangle \\ \mathbf{lock}(P \parallel R_0)\lceil\{a,b\} & = & \{\epsilon\} \\ L(P, R_0) & = & \mathbf{empty}(\{a,b\}) \end{array}$$

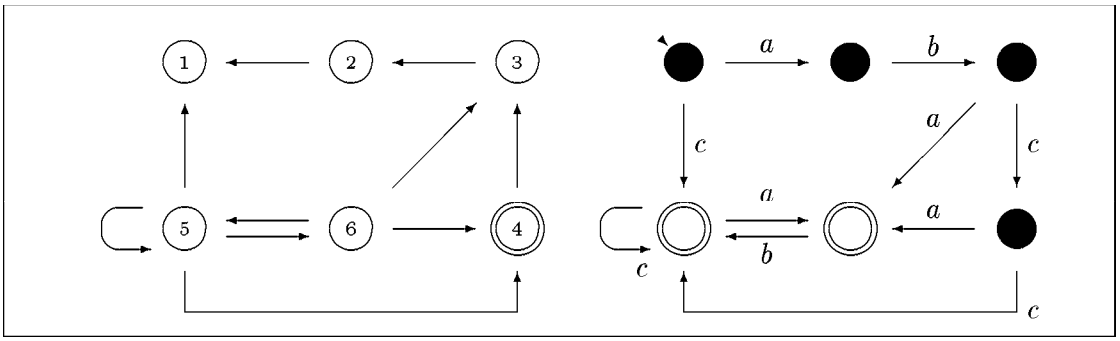
In fact we cannot control P using a and b such that no lock occurs.
(b) Computations:

$$\begin{array}{lcl} R_0 & = & \{a,c\}, (a|c)^* \rangle \\ \mathbf{lock}(P \parallel R_0)\lceil\{a,c\} & = & \{c\} \\ L(P, R_0) & = & \langle \{a,c\}, \{a,c\}, a^* \rangle \\ P \parallel R_0 & = & \langle \{a,b,c\}, ab \rangle \end{array}$$

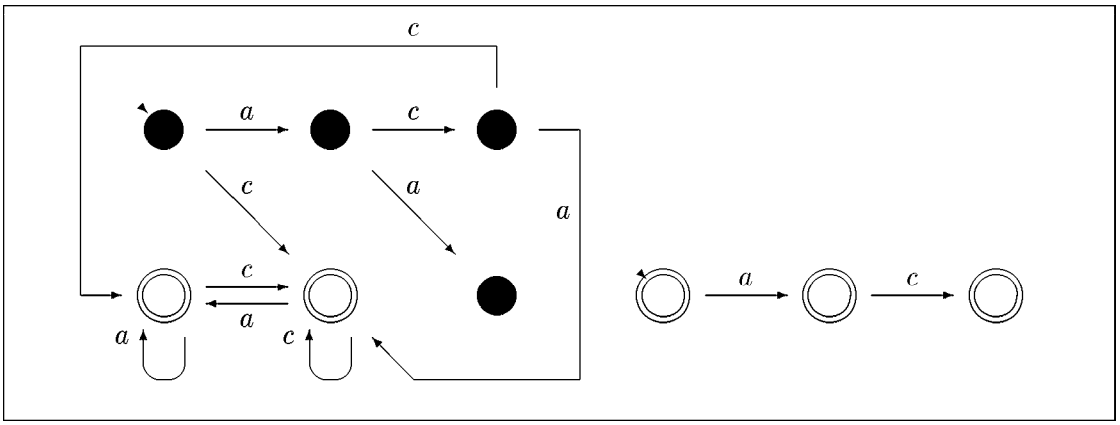
and is free of lock.
(c) Similar to previous one.

4.1 $\mathbf{sg}(P)$, $\mathbf{sg}(T)$ and $\mathbf{sg}(P)\backslash\mathbf{sg}(T)$ can be found in figure A.9.

4.2 $\mathbf{sg}(P)$ is given in figure A.10. Its reverse graph $\mathbf{rev}(P)$ is given in figure A.11. From that graph we see that states 5,6, and 7 (the dump state) are not reachable from state 4. So states 5 and 6 are lock states, which leads to the right graph in figure A.11 representing $\mathbf{lock}(P)$.



Figuur A.11: $\mathbf{rev}(P)$ *and* $\mathbf{sg}(\mathbf{lock}(P))$ *for exercise 4.2*



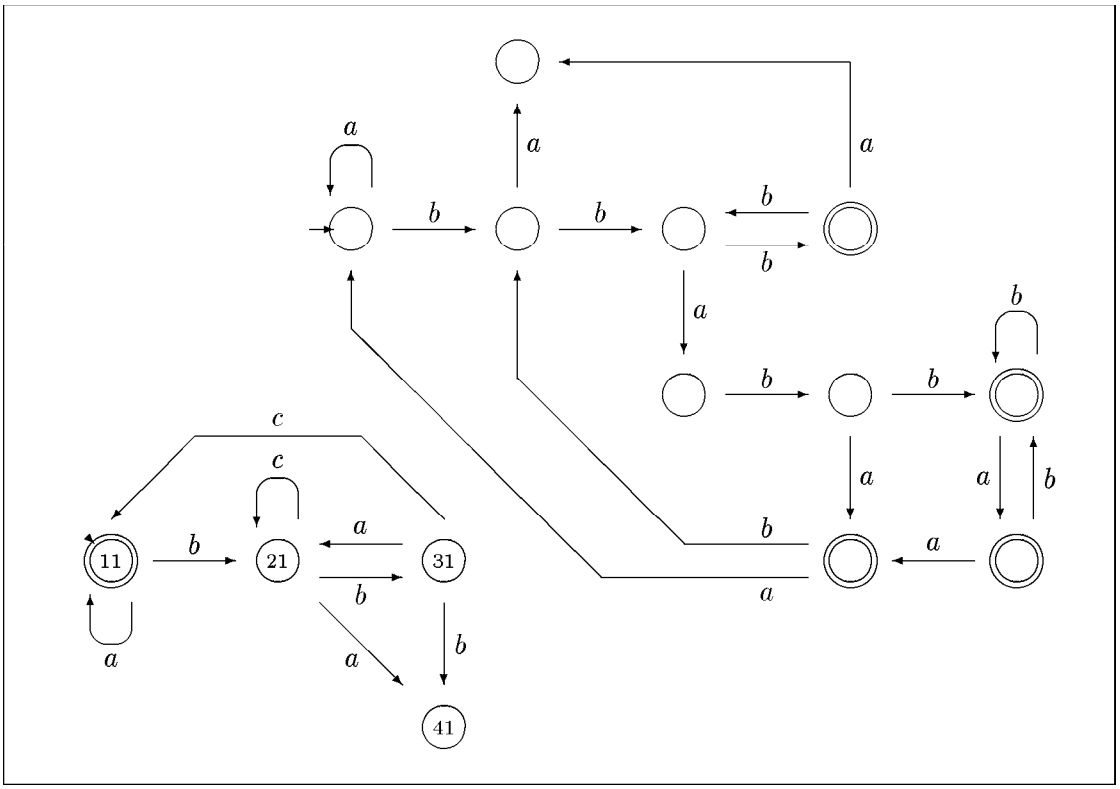
Figuur A.12: Graph for $\mathbf{lock}(P \parallel R_0)\lceil\mathbf{a}R)$ *(left) and* R_1 *(right) for exercise 4.3*

4.3 We use the same method as in the previous chapter: first start with R as large as possible, i.e., $R_0 = \langle\{a, c\}, (a|c)^*\rangle$.
First we compute $\mathbf{lock}(P \parallel R_0) = \mathbf{lock}(P)$ (see figure A.11). Next we compute $\mathbf{lock}(P)\lceil\mathbf{a}R$. The resulting graph $G = \mathbf{det}(\mathbf{sg}(\mathbf{lock}(P \parallel R_o)\lceil\mathbf{a}R))$ is given in figure A.12

4.4 We first compute $\mathbf{lock}(P \parallel R_0)$ with $R_0 = R$ and find the corresponding graph as given in figure A.13. We find a deadlock state: number 41.
The connection $P \| R$ (see figure 4.14 (c)) leads to deadlock in state 41. Computing $\mathbf{lock}(P \parallel R)\lceil\mathbf{a}R$ leads to the graph G_1 as is displayed in figure A.13.
Computing $R_1 = R_0 \setminus\setminus \mathbf{lock}(P \parallel R_0)\lceil\mathbf{a}R$ leads to the graph as is displayed in figure A.14. The connection $P \parallel R_1$ has (again) possibility to lock: state 25 is a livelock state. Notice that $P \parallel R_0$ contains a deadlock state while $P \parallel R_1$ contains a livelock state.
$P \parallel R_2$ contains a livelock state, namely state 24. It turns out that R_3 is the fix-point: all lock has disappeared.

5.1 First part:

$$
\begin{aligned}
& P \subseteq R \\
\Leftrightarrow\ & \mathbf{a}P = \mathbf{a}R \ \wedge\ \mathbf{b}P \subseteq \mathbf{b}R \ \wedge\ \mathbf{t}P \subseteq \mathbf{t}R \\
\Leftrightarrow\ & \mathbf{a}P = \mathbf{a}R \ \wedge\ \mathbf{b}P \cap ((\mathbf{a}R)^* \setminus \mathbf{b}R) = \emptyset \ \wedge\ \mathbf{t}P \cap ((\mathbf{a}R)^* \setminus \mathbf{t}R) = \emptyset
\end{aligned}
$$



Figuur A.13: Graphs for $P \parallel R_0$ (left) and $\mathbf{lock}(P \parallel R)|\mathbf{a}R$ *(right) for exercise 4.4*

$$\begin{aligned}
&\Leftrightarrow\ \mathbf{a}P = \mathbf{a}(\sim R)\ \wedge\ \mathbf{b}P \cap \mathbf{b}(\sim R) = \emptyset\ \wedge\ \mathbf{t}P \cap \mathbf{t}(\sim R) = \emptyset \\
&\Leftrightarrow\ P \parallel \sim R = \langle \mathbf{a}P, \emptyset, \emptyset \rangle
\end{aligned}$$

Second part: lhs. can only be true if $\mathbf{a}P \cup \mathbf{a}R = \mathbf{a}S$, rhs. can only be true if $\mathbf{a}R = \mathbf{a}P \cup \mathbf{a}S$. We also have ($\mathbf{a}R \subseteq \mathbf{a}P$ and $\mathbf{a}S \subseteq \mathbf{a}P$, which leads to

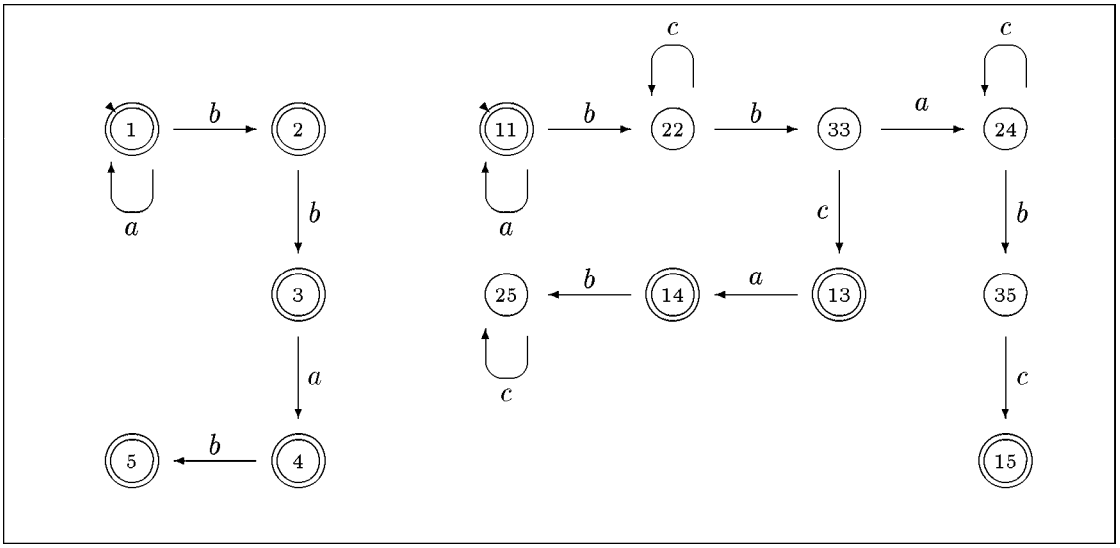
$$\mathbf{a}R = \mathbf{a}P = \mathbf{a}P$$

Under this condition we have:

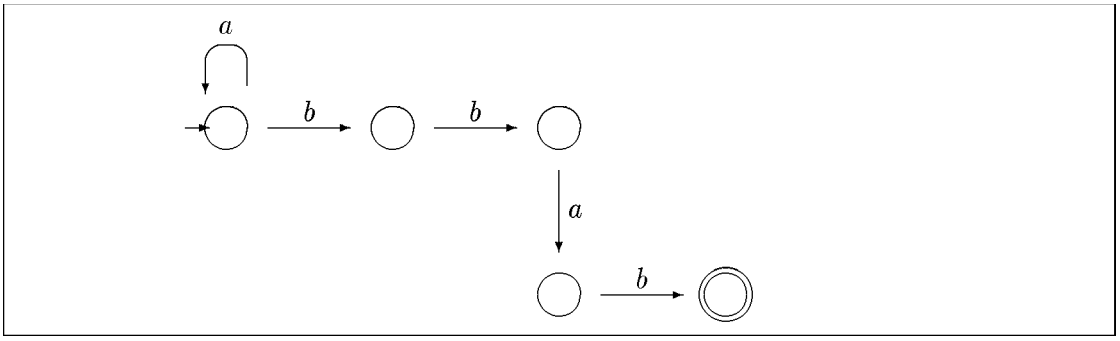
$$\begin{aligned}
&P \parallel R \subseteq S\ \wedge\ \mathbf{a}R = \mathbf{a}P = \mathbf{a}P \\
&\Leftrightarrow\ R \parallel P \subseteq S \\
&\Leftrightarrow\ (R \parallel P) \parallel \sim S = \mathbf{empty}(\mathbf{a}P)\ \wedge\ \mathbf{a}R = \mathbf{a}P = \mathbf{a}P \\
&\Leftrightarrow\ R \parallel (P \parallel \sim S) = \mathbf{empty}(\mathbf{a}P)\ \wedge\ \mathbf{a}R = \mathbf{a}P = \mathbf{a}P \\
&\Leftrightarrow\ R \parallel \sim\sim(P \parallel \sim S) = \mathbf{empty}(\mathbf{a}P)\ \wedge\ \mathbf{a}R = \mathbf{a}P = \mathbf{a}P \\
&\Leftrightarrow\ R \subseteq \sim(P \,]\!|\, \sim S)\ \wedge\ \mathbf{a}R = \mathbf{a}P = \mathbf{a}P
\end{aligned}$$

5.2 See figure A.19 for the state graphs of P and $L_{\max}$. We start with $L_{\max}$ and compute $\sim L_{\max}$, $P \,]\!|\, \sim L_{\max}$, and, finally $F(P, L_{\max})$ (see figure A.20). The greatest realistic solution is:

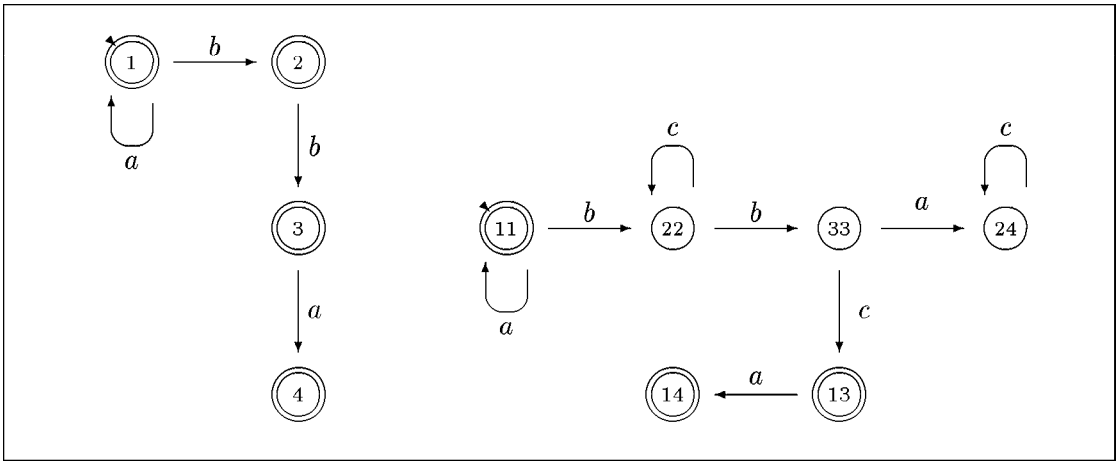
$$\mathbf{real}(F(P, L_{\max})) = \langle \{b, c, d\}, (b|c)(b|c|d)^* \rangle$$



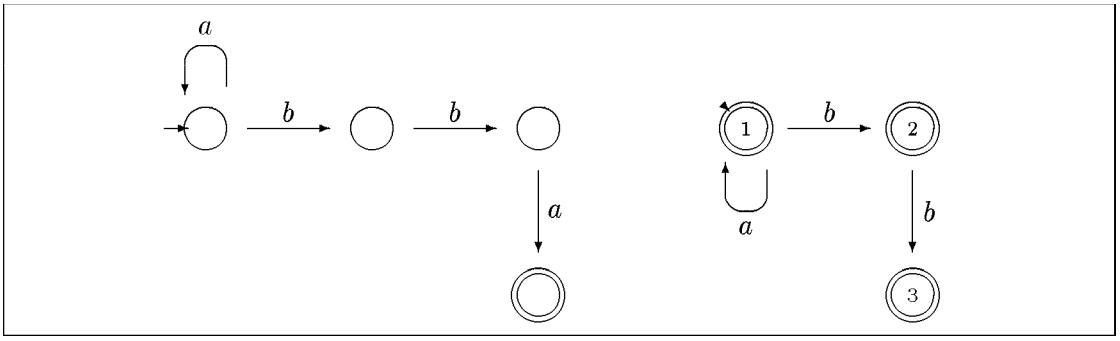
Figuur A.14: Graphs for R_1 (left) and $P \parallel R_1$ (right) with livelock in state 25.



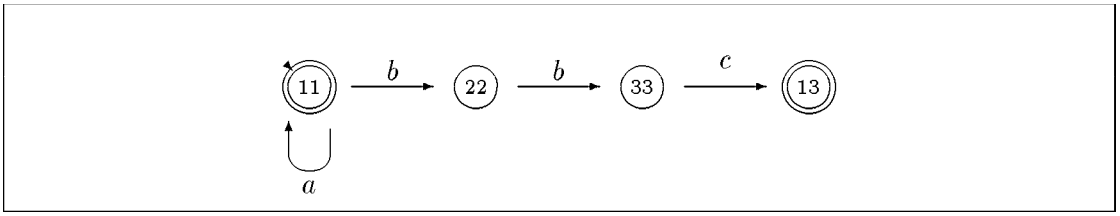
Figuur A.15: $\mathbf{lock}(P \parallel R_1)\lceil \mathbf{a}R$



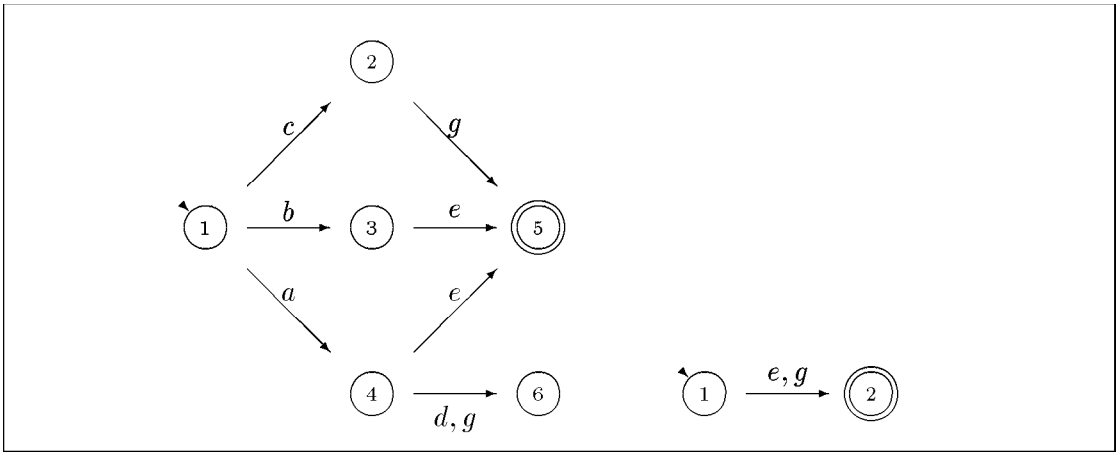
Figuur A.16: Graphs for R_2 (left) and $P \parallel R_2$ (right) with livelock in state 24.



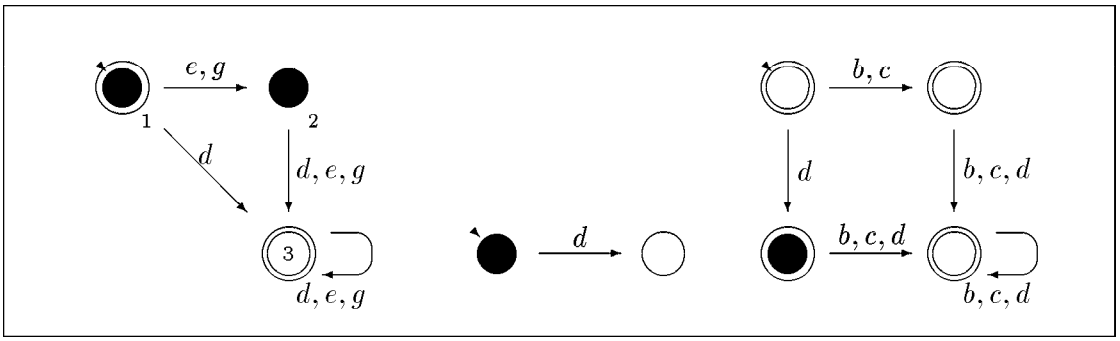
Figuur A.17: Graph for $\mathbf{lock}(P \parallel R_2)\lceil \mathbf{a}R$ *(left) and* R_3 *(right).*



Figuur A.18: $P \parallel R_3$ *which is free of lock.*



Figuur A.19: Graphs for P *and* $L_{\max}$ *for exercise 5.2*



Figuur A.20: Graphs for $\sim L_{\max}$ *(left),* $P \| \sim L_{\max}$ *(middle), and* $F(P, L_{\max})$ *(right) as constructed in exercise 5.2*

Figuur A.21: Systems L_1 (left) and L_2 (right) for exercise 6.3

5.3 We have:

$$\mathbf{t}(P \,[\!|\, R_1) = \{e\}$$
$$\mathbf{t}(P \,]\![\, R_2) = \{e\}$$
$$\mathbf{t}(P \,]\![\, (R_1 \cap R_2)) = \emptyset \not\supseteq \mathbf{t}L_{\min}$$

so both R_1 and R_2 are solutions of CODE, but $R_1 \cap R_2$ is not a solution. This is due to the fact that (according to property 1.23 (e)):

$$P \,]\![\, (R_1 \cap R_2) \subseteq (P \,]\![\, R_1) \cap (P \,]\![\, R_2)$$

In general, equality does not hold.

5.4 (a) is not observable, e.g., we have that $x = ae$ and $y = be$ both satisfy $x\lceil\mathbf{e}P = y\lceil\mathbf{e}P$ but $x\lceil\mathbf{c}P \neq y\lceil\mathbf{c}P$.
(b) is observable. Smallest solution for CODE is $R = \langle\{a, b\}, b\rangle$.
(c) is also observable. It has no solution for CODE.

5.5 Applying the given construction yields:

$$\begin{array}{lcl} P' & = & \langle\{a,b,c,d,e,g\}, \mathbf{pref}(ae|ad|ag|be|cg), (ae|be|cg)\rangle \\ \mathbf{e}P' & = & \{d,e,g,c\} \\ E & = & \{d,e,g\} \\ L^e_{\min} & = & L_{\min} \,\|\, P\lceil\mathbf{e}P = \langle\{d,e,g,c\}, e\rangle \\ L^e_{\max} & = & L_{\max} \,\|\, P\lceil\mathbf{e}P = \langle\{d,e,g,c\}, \mathbf{pref}(e|g|cg), (e|cg)\rangle \end{array}$$

This leads to

$$\begin{array}{lcl} F(P, L_{\max}) & = & \langle\{a,b\}, b(a|b)^*\rangle \\ P \,\|\, F(P, L_{\max}) & = & \langle\{a,b,c,d,e,g\}, (be|cg)\rangle \\ (P \,\|\, F(P, L_{\max}))\lceil E & = & \langle\{d,e,g\}, (e|g)\rangle \end{array}$$

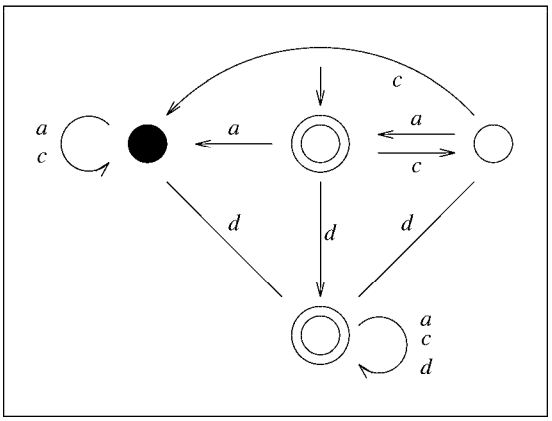
So indeed $R = \langle\{a, b\}, b(a|b)^*\}$ is the greatest solution.

5.6 From exercise 5.1 we derive that the construction $\sim(P \,\|\, \sim L_{\max})$ can only be used if $\mathbf{a}P = \mathbf{a}R$.

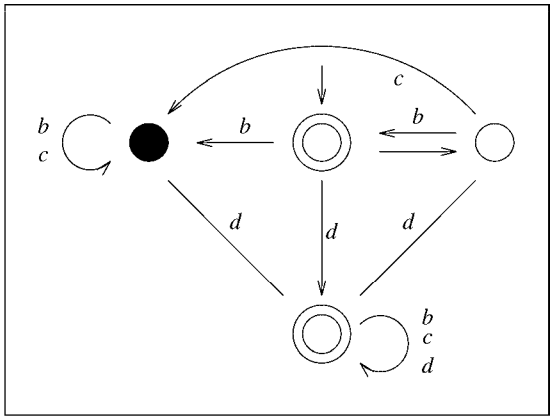
6.1 $\mathbf{c}P_i \cap \mathbf{e}P_i = \mathbf{c}P_i \cap (\mathbf{a}P_i \backslash \mathbf{c}P_i = \emptyset$
For $i \neq j$: $\mathbf{c}P_i \cap \mathbf{c}P_j = (\mathbf{c}P_i \cap \mathbf{a}P_i) \cap (\mathbf{c}P_j \cap \mathbf{a}P_j) \subseteq \mathbf{c}P_i \cap a_{ij} = \emptyset$
$a_{ij} = \mathbf{a}P_i \cap \mathbf{a}P_j = \mathbf{a}P_j \cap \mathbf{a}P_i = a_{ji}$

6.2 L is separable: for example take $L_1 = \langle\{a, b, c\}, ab\rangle$ and $L_2 = \langle\{b, c\}, b\rangle$. The system L' however is not separable w.r.t. the same alphabets.

6.3 The smallest possible generating set for L is $(L\lceil A_1, L\lceil A_2)$ with ($L\lceil A_1$ and $L\lceil A_2$ as in figure A.21.
Then we can compute $L_{11} = Q_1(L_1, L_2)$, which equals the system in figure A.22 and next compute $L_{22} = Q_2(L_{11}, L_2)$. It turns out that $L_{22} = L_2$, so every possible extension to L_2 is collected in the extended version of L_1.
We can also compute $L'_{22} = Q_2(L_1, L_2)$ first (resulting in the system of figure A.23) and computing $L'_{11} = Q_1(L_1, L'_{22})$ which leads to $L'_{11} = L_1$.



Figuur A.22: System L_{11} for exercise 6.3



Figuur A.23: System L'_{22} for exercise 6.3

Referenties

[AU72] A.V. Aho and J.D. Ullman.
The theory of parsing, translation and compiling, volume 1, parsing, volume 2, compiling.
Prentice Hall series in automatic computation, 1972.

[BCOQ92] F. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat.
Synchronization and Linearity.
Wiley, New York, 1992.

[BKS93] S. Balemi, P. Kozák, and R. Smedinga, editors.
Discrete Event Systems: Modeling and Control, volume 13 of *Progress in Systems and Control Theory*. Birkhäuser Verlag, Basel, Switzerland, 1993.
(Proceedings of the Joint Workshop on Discrete Event Systems (WODES'92), August 26–28, 1992, Prague, Czechoslovakia).

[BSW69] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson.
A note on reliable full-duplex transmissions over half-duplex links.
Communications of the ACM, 12 (5):260–261, 1969.

[Cas93] C.G. Cassandras.
Discrete Event Systems: Modeling and Performance Analysis.
Richadr D. Irwin, Inc., and Aksen Associates, Inc., Homewood, IL, USA, 1993.

[CDFV88] R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya.
Supervisory control of discrete event processes with partial observations.
IEEE transactions on automata and control, 33:249–260, 1988.

[CDQV83] G. Cohen, D. Dubois, J.P. Quadrat, and M. Viot.
A linear-system-theoretic view of discrete-event processes.
In *Proceedings of 22nd IEEE conference on decision control.* IEEE-press, Piscataway, 1983.

[CDQV85] G. Cohen, D. Dubois, J.P. Quadrat, and M. Viot.
A linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing.
IEEE transactions on automata and control, 30, 1985.

[Dij82] E.W. Dijkstra.
Predicate transformers.
lecture notes EWD835, Eindhoven university of technology, 1982.

[HC83] Y.C. Ho and X. Cao.
Pertubation analysis and optimization of queueing networks.
Journal on Opt. Th. Appl., 40, 1983.

[HC85] Y.C. Ho and C.G. Cassandras.
A new approach to the analysis of discrete event dynamic systems.
Automatica, 19, 1985.

[HEC83] Y.C. Ho, M.A. Eyler, and T.T. Chien.
A new approach to determine parameter sensitivities of transfer lines.
Management science, 29, 1983.

[Hes90] Wim H. Hesselink.
Modalities of nondeterminism.
In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beaty is our business, a birthday salute to Edsger W. Dijkstra*. Springer Verlag, 1990.

[Hoa85] C.A.R. Hoare.
Communicating sequential processes.
International series in computer science. Prentice Hall, 1985.

[HU79] J.E. Hopcroft and J.D. Ullman.
Introduction to automata theory, languages, and computation.
Addison Wesley, 1979.

[IV88] K. Inan and P. Varaiya.
Finitely recursive process models for discrete event systems.
IEEE transactions on automata and control, 33, 1988.

[Kal88] A. Kaldewaij.
A formalism for concurrent processes.
PhD thesis, department of mathematics and computing science, Eindhoven university of technology, 1988.

[LW88] Yong Li and W.M. Wonham.
Deadlock issues in supervisory control of discrete event systems.
In *Proceedings of the 1988 conference on information science and systems.* Princeton University, Princeton NJ, 1988.

[Mil80] R. Milner.
A calculus for communicating systems.
Lecture notes in computer science, nr. 92. Springer Verlag, 1980.

[Pet81] J.L. Peterson.
Petri Net theory and the modelling of systems.
Prentice-Hall, 1981.

[Ram83] P.J. Ramadge.
Control and supervision of discrete event processes.
PhD thesis, systems and control group, department of electrical engineering, University of Toronto, 1983.

[RW87] P.J. Ramadge and W.M. Wonham.
Supervisory control of a class of discrete event processes.
SIAM journal on Control and optimalisation, 25 (1), 1987.
See also: systems control group report 8515, department of electrical engeneering, University of Toronto.

[RW89] P.J. Ramadge and W.M. Wonham.
The control of discrete event systems.
Proceedings of the IEEE, 77(1), 1989.

[RW92] Karen Rudie and W.Murray Wonham.
Think globally, act locally: decentralized supervisory control.
IEEE transactions on automatic control, 37(11), november 1992.

[Sme88] R. Smedinga.
Using trace theory to model discrete events.
In P. Varaiya and A.B. Kurzhanski, editors, *Discrete event systems: models and applications*, Lecture notes in control and information science nr. 103, Workshop Sopron, Hungary, August 3–7 1988. IIASA, Springer Verlag.

[Sme89] R. Smedinga.
Control of discrete events.
PhD thesis, University of Groningen, 1989.

[Sme90a] R. Smedinga.
Locked discrete event systems.
Technical Report CS9002, Department of computing science, University of Groningen, 1990.

[Sme90b] R. Smedinga.
More about locked discrete event systems.
Technical report, Department of computing science, University of Groningen, 1990.
in preparation.

[Sme91] R. Smedinga.
Discrete event systems: deadlock, livelock, and livedeadlock.
In U. Jaaksoo and V.I. Utkin, editors, *Automatic Control, world congress 1990, 13–17 August, proceedings of 11th IFAC world congress*, volume III, Tallinn, Estonia, USSR, 1991. Pergamon Press.

[Sme92] R. Smedinga.
The reflection operator in discrete event systems.
Technical Report CS9201, Department of computing science, University of Groningen, 1992.

[Sme93a] R. Smedinga.
Letter to the editor.
Journal on Discrete Event Dynamic Systems, theory and applications, 3(4):317–321, 1993.

[Sme93b] R. Smedinga.
Locked discrete event systems: how to model and how to unlock.
Journal on Discrete Event Dynamic Systems, theory and applications, 2(3/4), 1993.

[Sme93c] R. Smedinga.
An Overview of Results in Discrete Event Systems using a Trace Theory Based Setting, pages 43–56.
Volume 13 of Balemi et al. [BKS93], 1993.
(Proceedings of the Joint Workshop on Discrete Event Systems (WODES'92), August 26–28, 1992, Prague, Czechoslovakia).

[Sme93d] R. Smedinga.
The Workshop Exercise Using a Trace Theory Based Setting, pages 167–172.
Volume 13 of Balemi et al. [BKS93], 1993.
(Proceedings of the Joint Workshop on Discrete Event Systems (WODES'92), August 26–28, 1992, Prague, Czechoslovakia).

[Sne85] J.L.A. van de Snepscheut.
Trace theory and VLSI design.
Lecture notes in computer science, nr. 200. Springer Verlag, 1985.

[T.V90] T.Verhoeff.
Solving a control problem.
Internal report, Eindhoven University, 1990.

[T.V91] T.Verhoeff.
Factorization in process domains.
Internal report, Eindhoven University, 1991.

[TW86] J.G. Thistle and W.M. Wonham.
Control problems in a temporal logic framework.
International Journal on Control, 44, 1986.
See also: Systems and control group report 8510, department of electrical engineering, University of Toronto.

[Udd84] J.T. Udding.
Classification and composition of delay-insensitive circuits.
PhD thesis, department of mathematics and computing science, Eindhoven university of technology, 1984.

[vG88] A.J.M. van Gasteren.
On the shape of mathematical arguments.
PhD thesis, University of Eindhoven, 1988.

[Wil88] Jan C. Willems.
Models for dynamicals.
In U. Kirchgraber and H.O. Walther, editors, *Dynamics Reported, volume 2*, chapter 5. John Wiley & Sons, 1988.

[WM91] Y. Willner and M.Heymann.
Supervisory control of concurrent discrete-event systems.
International journal on control, 54(5):1143–1169, 1991.

[WR87] W.M. Wonham and P.J. Ramadge.
On the supremal controllable sublanguage of a given language.
SIAM journal on control and optimalisation, 25 (3), 1987.

Index

Glossary

chapter 1

$x \cdot y$	concatenation of traces x and y	5
xy	abbreviation for $x \cdot y$	5
$x \mid y$	choice between x and y	5
x^n	finite repetition of trace x (n times)	6
x^*	repetition of x (zero or more times)	6
x^+	non-zero repetition of x	6
x, y	weaving of x and y	6
$x \lceil A$	projection of x on alphabet A	6
$x \lceil A$	trace structure x restricted to alphabet A	6
$\mathbf{pref}(x)$	prefix closure of trace x	6
$\mathbf{pref}(T)$	prefix closure of trace set T	6
A^*	alphabet generating set	6
$x \leq y$	x is prefix of y	6
$x < y$	x is prefix of y and unequal to y	6
$\|x\|$	length of trace x	6
$x \mathbf{N} a$	number of occurrences of event a in trace x	6
$\langle \mathbf{a}P, \mathbf{b}P, \mathbf{t}P \rangle$	discrete event system with alphabet $\mathbf{a}P$, behaviour set $\mathbf{b}P$ and task set $\mathbf{t}P$	9
$\mathbf{empty}(A)$	the empty system	9

continues

$\mathbf{skip}(A)$	the skip system	9
$\mathbf{chaos}(A)$	the chaos system	9
$P \subseteq R$	ordering on systems	10
$P\lceil A$	restriction of P to alphabet A	10
$\mathbf{real}(P)$	the realistic interior of P	10
$\sim P$	dual system of P	11
$P \parallel R$	connection of P and R	13
$P \rceil\!\parallel R$	external connection of P and R	15
$P \cup R$	union of P and R	16
$P \cap R$	intersection of P and R	16
$P \backslash R$	P without R	16
$(\parallel P : P \in \mathcal{P} : P)$	connection of set of systems	18
$a!$	output event a	19
$a?$	input event a	19
$\mathbf{o}P$	output alphabet of P	19
$\mathbf{i}P$	input alphabet of P	19
$P?!$	directed DES	19
$P \overset{\leftrightarrow}{\parallel} R$	directed connection of P and R	19

chapter 2

(A, Q, δ, q, B, T)	state graph	23
δ^*	closure of transition function δ	24
$[x]_P$	equivalence class of trace x in P	24
$\mathbf{sg}(P)$	state graph of system P	24
$\mathbf{S}(P)$	state set of $\mathbf{sg}(P)$	24
$\mathbf{q}(P)$	initial state of $\mathbf{sg}(P)$	24
$\mathbf{B}(P)$	behaviour state set of $\mathbf{sg}(P)$	24
$\mathbf{T}(P)$	task state set of $\mathbf{sg}(P)$	24
$\mathbf{des}(G)$	system represented by state graph G	25
$G_{\mathrm{empty}}(A)$	state graph for $\mathbf{empty}(A)$	26
$G_{\mathrm{skip}}(A)$	state graph for $\mathbf{skip}(A)$	26
$G_{\mathrm{chaos}}(A)$	state graph for $\mathbf{chaos}(A)$	26

continues

$\mathbf{reachable}(G)$	reachable subgraph of G	26
$(A, Q, \gamma, q, B, F)_{\mathrm{nd}}$	nondeterministic state graph	29
$\mathbf{det}(G_{\mathrm{nd}})$	deterministic equivalence of nondeterministic graph	30
$G\lceil A$	restriction of graph G to alphabet A	32
$G_1 \parallel G_2$	connection of state graphs G_1 and G_2	34
$\sim G$	reflection of state graph G	37
$\mathbf{real}(G)$	des-interior of state graph G	38
$G_1 \cup G_2$	union of graphs G_1 and G_2	39
$G_1 \cap G_2$	intersection of graphs G_1 and G_2	39
$G_1 \backslash G_2$	exclusion of graphs G_1 and G_2	39

chapter 3

$\mathbf{deadlock}(P)$	deadlock traces of P	43
$\mathbf{deadlockfree}(P)$	$\mathbf{deadlock}(P) = \emptyset$	43
$\mathbf{livelock}(P)$	livelock traces of P	44
$\mathbf{lock}(P)$	locked traces of P	45
$\mathbf{lockfree}(P)$	$\mathbf{lock}(P) = \emptyset$	45
$P \setminus\setminus T$	P without traces T	46
$L(P, R)$	operator to find the greatest possible subsystem of R that leads to a lockfree connection with P	47
$\Lambda(P, R)$	greatest subsystem of R that leads to lockfree connection with P	51
$\Delta(P, R)$	greatest subsystem of R that leads to a deadlock free connection with P	53

chapter 4

chapter 5

$\mathbf{e}P$	exogenous alphabet of P	68
$\mathbf{c}P$	communication alphabet of P	68
$F(P, L)$	friend of P and L	70

continues

$G(P, L)$	guardian of P and L	70
$G(P, L)$	resulting external connecting of P and $F(P, L)$	70
$F_{\text{real}}(P, L)$	realistic interior of $F(P, L)$	72
$G_{\text{real}}(P, L)$	guardian for $F_{\text{real}}(P, L)$	72
$\textbf{observable}_E(P)$	P is observable with respect to exogenous events E	76
$E(P, L)$	extended friend of P and L	86
$H(P, L)$	host of P and L	86

chapter 6

$\mathbf{c}_i P$	partial set of controls for one local controller in GsLc	90
$\mathbf{a}_{ij}$	cooperating evens for P_i and P_j for LsLc	91
$(L_1, \ldots, L_n)$	generating set	100
$B(L)$	set of all generating sets for L	100
$B_i(L)$	set of all i-th components of generating sets of L	100
$Q_i(L_1, \ldots, L_n)$	returns largest i-th component for generating set $(L_1, \ldots, L_n)$	103